冲呀程序员系列



程序员的三门课

技术精进、架构修炼、管理探秘

于君泽　李伟山　张洪亮　彭首長　刘　朋　著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

我们都有一个共同的梦想——成为更棒的程序员，但是如何做？如何学习和精进自己的技术？如何做业务分析和架构设计？如何做技术管理？本书就广大程序员都很关注的问题提供一些思路和方法。

本书内容分为三大篇：第 1 篇介绍程序员的技能成长路径，介绍如何通过学习来加速成长，并讲解非常重要的业务分析和设计知识；第 2 篇详细介绍了架构修炼中的必备架构思维模式、架构设计过程，还通过真实案例剖析了架构设计的核心要素及关注点，以及如何通过架构设计来把控质量和风险；第 3 篇总结了做技术管理的诀窍，揭秘如何从准备做技术管理到实现自我管理再到实现团队管理。

本书适用于即将或者正在从事 IT 相关开发工作，却不知如何提升自己，希望通过适当的学习方法精进技术的初级程序员；也适用于具备一定架构设计经验，想进一步系统化提升架构设计能力的中级程序员；以及有一定技术积累，想从事技术管理却不知如何着手的高级程序员。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

程序员的三门课：技术精进、架构修炼、管理探秘 / 于君泽等著. —北京：电子工业出版社，2019.10
（冲呀程序员系列）
ISBN 978-7-121-37377-0

Ⅰ. ①程… Ⅱ. ①于… Ⅲ. ①程序设计 Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字（2019）第 194241 号

责任编辑：张国霞
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编 100036
开　　本：787×980　1/16　印张：18.5　字数：390 千字
版　　次：2019 年 10 月第 1 版
印　　次：2019 年 10 月第 1 次印刷
印　　数：4000 册　定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

推荐序 1

世界需要什么样的程序员

不想当工程师的程序员不是好民工。会编程不代表会工作，工程能力包含软硬技能，面向的是实际应用。从程序员变为工程师，得先“入门”，然后才是“上道”，从而成为“工型人才”。有人工作几年还没入门，只会简单开发；有人开发技能娴熟，却找不到进一步成长的路径，陷入迷茫之中；更有不少人自诩为 IT 精英，却不了解行业格局、发展趋势，很快就遇到职业发展瓶颈，蹉跎了岁月，虚度了青春。

在我看来，好的程序员应该是“工型人才”。所谓“工型”，是指从下到上的能力提升过程。具体来说，就是先要具备完成完整应用的能力，包括：线上运维，成为熟手，这是下面的一横；在某些领域足够深入，成为高手，这是中间的一竖；在达到更高的水平之后，兼通很多领域，比如业务、产品、项目管理、测试、运维、团队组织，成为驱动者和领导者，这是上面的一横。

每个人的成长都是不断打怪升级的过程，既要掌握技术和工具，又要学习方法和理论、积累实战经验，更要思考和沉淀。先进技术只是工具，最终目标是解决问题。经验丰富的工程师就像 MMA（Mixed Martial Arts，综合格斗）高手，精通综合技巧，活学活用，不拘泥于流派。在勇猛、精进之外，踏实、靠谱更为重要。踏实，是有自己的方向，知道差距；靠谱，是能把活儿干好，值得大家信赖，大家都乐于和你合作，放心把有挑战的任务交给你。

从程序员到工程师，读这本书就够了！本书非常实用，可称作程序员入行手册。市场上的各类“21 天学会 XXX”和“XXX 编程宝典”书已经太多了，能够传承工作中实践技巧的书依然太少。本书汇聚了多位资深技术老炮儿压箱底的傍身秘技，兼顾软硬技能，贯通了程序员从基础到进阶再到突破的各个阶段，道出了成长的真谛，更手把手地指引我们前行。

有道是“师傅领进门，修行在个人”，徘徊在门外或者迷茫在路口的新同学，从此不必再上下求索、苦苦寻觅。一书在手，夫复何求？希望你能通过本书，提升程序员的大局观，在成为“工型人才”的路上迈出坚实的一步，并穿越众妙之门，通达三千世界。

微信公众号“IT 民工闲话”作者　史海峰

推荐序 2

程序员的工作是将从现实或者虚拟世界中抽象出来的逻辑以代码的形式实现，需要凝神静气才能完成一段有质量的代码。但不得不承认的是，程序员这个职业在很大程度上是吃青春饭的，如何充分把握自己的青春年华做一名优秀程序员？如何为未来的职业发展打下良好基础？这是每个程序员都应该认真思考的问题。

非常遗憾的是，程序员这个职业所需要的知识、技能和经验，并不能完全从学校获得，更多地来自工作实践。本书系统且循序渐进地介绍了如何从菜鸟程序员成为高级研发人员或者架构师，以及如何积累知识和经验成为技术管理者，对于程序员在编码过程中需要用到的工具、模式、测试、沟通和架构知识，作者也给出了亲身实战经验总结。

本书应该成为程序员的案头手册，对程序员的快速成长意义重大。读了这本书，相当于同老一代程序员深入交谈，他们也把自己的酸甜苦辣、成功与挫折、心得与秘密悄悄地告诉了你。这不仅仅建立了你的大局观，也让你有机会有赢在起跑线上。

易宝支付 陈斌

序1

本书的素材主要来源于我的多年公众号积累，并邀请了李伟山、张洪亮、彭首長、刘朋这几位专家共同完成本书的写作，杨彪老师也在本书写作前期贡献了不少想法。

我一直都是一个“笨孩子”，但我不以笨为耻。小学三年级时要写一篇关于柑橘树的作文，那时自己肚子没货、下笔万难，父亲知道后教了我两招：一是从整体到局部观察事物；二是阅读，看看别人是怎样写的。

有朋友问，写书的时间从哪里来？其实，因为热爱，不觉疲惫！写作是让自己梳理思路的一个机会。古人说，有三类人，一类人深沉厚重，一类人磊落豪情、一类人聪明才辩。聪明才辩是学不了了，磊落豪情在骨子里有几分，愿学深沉厚重，下笨功夫。

中国的软件业发展短短几十年，在思想构建、方法体系乃至实践上仍有很多需要我们努力的地方，每一年都有人在问十年前就被问过的问题，说明有些路必须走过才知道，也说明如果能在走路的过程中给后来者一点点益处，也算是一桩好事！

特别感谢自己读高中时的班主任何老师，何老师教给我了独立思维，教我在写作功力扎实的基础上探索更有新意的内容。有一次作文课，何老师念了其他同学的两篇文章，课后跟我说到，其实我写得也不错，但也要给其他同学机会。我感受良多，也渐渐地学会了包容。

感谢同事、朋友的帮助，愿我们共同成长！

感谢我的妻子小娜对我的支持和帮助，感谢你不断鼓励和支持我去做新的尝试。我们的一对可爱的儿女也带给我很多欢乐。所谓幸福，大抵如此！谢谢你们！

于君泽（右军） 于成都

序2

为什么要写书呢？这要从我的切身体会说起，刚进入职场时自己的职业规划几乎为零，一心想着怎样尽快学习更多的技术和积累更多的业务经验，但现实是每天都深陷永无止尽的业务需求开发中。因为长期埋没于代码世界的浩大分工体系中，我无法看清从业务到系统架构的价值链条，无法清晰定义自己在分工体系中的位置，处理不好自身与技术、业务的关系。

我想，要是能有人帮我规划，我可能进步得更快。好在自己总能给自己打气（总觉得自己很棒），经过这么多年的摸爬滚打，终于摸索出相对清晰的职业发展路径。为了帮助和我一样致力于在 IT 道路上走得更远的人，我参与了本书的写作。

写书是一个重新梳理自己的知识体系的过程，也是重新梳理自己的逻辑思维的过程。写书和写公众号不一样，公众号只是分享一个知识点，而写书是分享一个相对完整的知识体系，需要思维逻辑更加缜密。当然，还要做好时间安排，我写书都是先在路上梳理思路，再回家写。

幸运的是，2018 年有了儿子嘟嘟，当嘟嘟好奇地想摸电脑时，我会暂停写书陪他，假模假样地给他讲爸爸在干什么。当然，儿子也是自己写书的动力。

感谢右军、刘朋、彭首長、张洪亮，很荣幸和你们一起合作，虽然素未谋面，但我从字里行间可以感受到你们对技术的热情和真知灼见，希望我们能有更多的合作。

最后，郑重感谢妻子小何同学为我们的家庭做出的贡献。

李伟山 于南京

序3

我在 2015 年离开校园，现在还依稀记得当初刚刚入职的时候，公司里的高管和我们新入职的员工一起交流，他说：“一个程序员工作的前 3 年，肯定是他成长最快的 3 年”。我当时对这句话的理解还不够深，但随着自己慢慢地走过第一个 3 年，回首过去，发现在这 3 年里，自己确实经历了一个飞速成长的过程。但是，自己也曾迷茫、困惑。所以，我特别想把自己在这 3 年里总结的经验分享给其他人。

其实我很早就开始写博客了，有很多出版社的编辑找过我，也有很多读者问我能不能写本书，但都被我婉拒了，因为我始终觉得自己还需要沉淀一段时间。直到 2018 年，在我工作满三年，正有向其他人分享自己的工作经验想法的时候，右军找到我，说他想出一本关注程序员成长的书。我们一拍即合，马上就投入到了这本书的创作当中。

在写这篇作者感悟的时候，我简单回顾了自己的人生，觉得自己还是挺幸运的。初中升高中时压线升入一所省重点，高中升大学时压线考入自己想学的专业，大学毕业时也幸运地加入自己梦寐已求的公司。我在工作的这几年里一直积极跟进技术的发展，所幸丝毫没有耽误自己的成长，而且能坚持更新博客和公众号，收获了很多读者和粉丝。这可能就是所谓的“越努力越幸运”吧。

特别感谢本书的发起人右军的邀请，并感谢本书的其他几位作者：李伟山、刘朋、彭首長，还要感谢本书的责任编辑张国霞，以及为本书的写作做出很多贡献的杨彪，很荣幸能和大家一起完成这样一本书的写作！

感谢我的博客和公众号上一直支持我的读者和粉丝，是你们的每一次阅读和每一条评论给了我坚持创作的动力！

感谢我的女朋友对我的理解与支持！

感谢我的爸爸妈妈在我的成长道路上给我的包容与支持！

张洪亮 于杭州

序4

在刚收到右军的本书写作邀请时，其实我是拒绝的，因为当时的 IT 圈遍布裁员和 996 的消息，几乎所有人都在认真对待自己的工作，希望以更多的输出稀释空气中的那份压抑，我也希望把更多的时间和精力留给工作。

但是，人生总是有那么多的偶然性，一位已步入中年的员工在离职之际向我倾诉了他的迷茫、焦虑和无助。他在经历了多年的职业生涯后还是没能找到自己的职业方向，将大量的时间消耗在了十字路口的彷徨中，也没能让自己高效成长。风雨之中，这位中年男人疲惫地带着梦想蹒跚前行。他的倾诉也让我陷入了深深的沉思中，总觉得自己应该做些什么。于是，我加入了本书的写作团队中，与其他作者一起，把我们在互联网行业里走过的路、踩过的坑和做过的思考以文字的形式呈现给大家，希望对正在路上的朋友有所帮助。

特别感谢本书的几位作者：右军、李伟山、张洪亮、刘朋，我们齐心协力完成了本书的写作，我在该过程中也受益颇多，在技术和认知上都得到了提升。感谢本书的责任编辑张国霞，以及为本书的写作提供过很多帮助的朋友，特别是那些在百忙之中帮我找错别字的小伙伴：何宇、潘晨、杨笑美、赵元舒，大家辛苦了！

彭首長 于成都

序5

在一堂培训课上，关于领导力，一位技术管理者说：“领导力就是心存善念、劝人向善”，大家都在短暂的沉默后，被这有点“佛系”的说法逗乐了。但我认为，“心存善念、劝人向善”这八个字，不仅适用于技术领导者，还适用于程序员的成长。

例如，在下班之后，把刷微信的时间用在为 GitHub 上的开源项目贡献一些整洁的代码。也许你只是希望自己在用了这么多年开源项目之后，能够回馈社区。但你没有想到，在自己做技术面试时，面试官突然抛开你的简历，关注起你在 GitHub 上的技术贡献。

又如，当团队中的大多数人只关注项目进度，忽略了新增的代码是否让整个系统的架构变得愈发复杂时，你出于团队不要被“腐烂”的代码拖慢进度的想法，自告奋勇地用刚学的“整洁代码”技能，小心翼翼地把系统做了一次重构。当大家都觉得你“另类”时，你却发现自己不经意间提升了架构的技能，也进入了老板的后备架构师人才库。

对我来说，能够出版自己的第 1 本书，是因为右军推荐我在 2016 年的 TOP 100 大会上做分享；能够参与出版这第 2 本书，是因为自己不断参与社区的活动，与其他贡献者在不经意间碰撞出火花。

感谢技术社区的小伙伴们，以及我的同事和朋友们，和你们在一起，是我最快乐的时光！感谢本书的责任编辑张国霞，她对我们的延迟交稿表现出足够的耐心，在多次不厌其烦地审阅和校对过程中，都展现出极高的专业性，为本书增色良多。

特别要感谢的，是我的妻子和父母，没有你们的爱、支持和理解，这本书是不可能完成的。在所有的风景当中，你们都是最美的！

刘朋 于成都

目录

第 1 篇　技术精进

第 1 章　程序员技能与成长 2

1.1　如何学习新的编程语言 2

1.1.1　重点学什么 3

1.1.2　学习方法 3

1.2　代码规范与单元测试 6

1.2.1　编码规范 6

1.2.2　单元测试 7

1.2.3　测试驱动设计 9

1.3　使用静态代码分析工具 18

1.3.1　什么是静态代码分析 19

1.3.2　静态代码分析工具 19

1.4　代码审查 24

1.4.1　什么是代码审查 24

1.4.2　代码审查的好处 24

1.4.3　如何做代码审查 25

1.5　清单和模板 29

1.5.1　清单实践 29

1.5.2　产品需求文档模板 31

1.6　程序员的工作法则 33

1.6.1　工具化法则 33

1.6.2 自动化法则 38
1.6.3 关于文档的问题 40
1.6.4 关于 YAGNI 40
1.7 程序员的工具箱 41
1.7.1 可帮我们提升工作效率的硬件 42
1.7.2 可帮我们提升工作效率的操作系统 43
1.7.3 可帮我们提升工作效率的软件 43
1.7.4 在线工具 50
第 2 章 加速成长与学会学习 56
2.1 程序员如何加速成长 56
2.1.1 积极主动 56
2.1.2 空杯心态 57
2.1.3 选择合适的平台 57
2.1.4 别怕犯错 58
2.1.5 注意细节 58
2.1.6 时间管理 58
2.1.7 打破边界 61
2.1.8 写业务代码中的成长机会 61
2.2 学会学习 66
2.2.1 高效学习的方法 67
2.2.2 高效学习的途径 69
2.2.3 舒适区与小步子原则 72
2.2.4 两个著名的学习理论 72
2.2.5 终身成长 74
第 3 章 业务分析与设计 75
3.1 黄金圈法则 75
3.2 UML 建模工具 77
3.2.1 用例图 78
3.2.2 类图 79
3.2.3 对象图 80
3.2.4 状态图 80

3.2.5 活动图 81
3.2.6 序列图 82
3.2.7 协作图 82
3.2.8 构件图 83
3.2.9 部署图 84
3.3 业务分析与设计的方法 85
3.4 系统分析与设计的三个发展阶段 87
3.4.1 面向数据驱动分析与设计 87
3.4.2 面向对象和服务分析与设计 88
3.4.3 面向问题域分析与设计 89
3.5 面向对象分析与设计 90
3.5.1 什么是面向对象 90
3.5.2 面向对象的特征 91
3.5.3 面向对象设计的原则 91
3.6 面向服务分析与设计 93
3.6.1 Web Service 模式 93
3.6.2 ESB 模式 94
3.6.3 微服务架构 95
3.7 领域驱动设计 97
3.8 领域模型实践 102
3.8.1 什么是领域模型 102
3.8.2 领域模型的作用 102
3.8.3 如何进行领域建模 104

第 2 篇 架构修炼

第 4 章 架构思维 110
4.1 分解 110
4.1.1 分解的作用 111
4.1.2 分解的原则 111
4.1.3 分解的时机 112

4.2 集成 112
4.2.1 常见的集成方式 112
4.2.2 集成的难点 115
4.3 动静分离 120
4.4 复用 121
4.4.1 系统复用的分类 122
4.4.2 可复用性和可维护性的关系 123
4.4.3 可维护性地复用的设计原则 123
4.5 分层 126
4.6 模式 127
4.6.1 分层模式 128
4.6.2 客户端-服务器模式 131
4.6.3 主从设备模式 132
4.6.4 管道-过滤器模式 133
4.6.5 代理模式 133
4.6.6 P2P 模式 134
4.6.7 事件总线模式 135
4.6.8 MVC 模式 136
4.6.9 黑板模式 137
4.6.10 解释器模式 138
4.7 抽象 139
4.8 结构化 139
4.8.1 结构化的原则 140
4.8.2 结构化分析工具 140
4.8.3 结构化思维的 7 个环节 141
4.9 迭代 141
4.10 勿做过度设计 141
第 5 章 架构设计 143
5.1 架构设计概要 143
5.1.1 业务架构 143

5.1.2 应用架构 145
5.1.3 技术架构 147
5.1.4 数据架构 149
5.2 架构设计的流程 151
5.3 架构设计的核心要素 153
5.3.1 性能 153
5.3.2 可用性 153
5.3.3 伸缩性 154
5.3.4 扩展性 154
5.3.5 安全性 154
5.4 高性能设计 154
5.4.1 主要的性能测试指标 155
5.4.2 性能测试方法 156
5.4.3 性能优化策略 157
5.5 高可用设计 159
5.5.1 高可用性的度量与考核 159
5.5.2 高可用的架构 160
5.5.3 高可用质量保证 164
5.5.4 系统运行监控 165
5.6 可伸缩设计 166
5.6.1 架构的伸缩性设计 166
5.6.2 应用服务的伸缩性设计 167
5.6.3 分布式缓存的伸缩性设计 169
5.6.4 数据存储的伸缩性设计 169
5.7 可扩展性设计 169
5.7.1 通过消息降低系统的耦合性 170
5.7.2 通过分布式服务提供可复用的业务 170
5.7.3 建立开放平台 170
5.8 架构设计的典型案例——订单系统 171
5.8.1 订单的基本概念 172
5.8.2 订单流程 174
5.8.3 订单系统设计中的挑战和实践 178

5.8.4 数据库优化 182
5.8.5 小结 185

第 6 章 架构的保障：质量与风险 186

6.1 内建质量体系 186
6.2 从黑天鹅事件到墨菲定律 189
6.2.1 黑天鹅事件 189
6.2.2 蝴蝶效应 191
6.2.3 墨菲定律 191
6.3 软件质量稳定性之殇 192
6.3.1 业务高速发展带来的变化 193
6.3.2 问题域的复杂性 193
6.3.3 系统的复杂性 194
6.3.4 技术债问题 194
6.3.5 人、流程、文档的博弈 195
6.3.6 采用不能掌控的工具和框架 196
6.3.7 质量意识 196
6.4 从康威定律和技术债看研发之痛 197
6.4.1 康威定律 197
6.4.2 通过 3 个案例来看研发之痛 197
6.4.3 架构方案和业务形态息息相关 204
6.5 求解质量熵 206
6.5.1 运用敏捷思想 207
6.5.2 运用系统化思想 208
6.5.3 技术债偿还计划 209
6.5.4 抓住合适的时机做架构升级 209
6.5.5 内建质量 210
6.5.6 不迷信，不唯新、不唯上，实践是检验真理的标准 210
6.5.7 复杂的问题域：专项突破 210
6.5.8 领导者的意识 211
6.5.9 创新解决方案 211

6.6 踩过的坑和经验总结 212
6.6.1 简单的数字改动导致电商网站搜索功能失灵 212
6.6.2 应用竟然被外部系统拖垮了 212
6.6.3 HashMap 与并发 213
6.7 故障复盘流程及模板 215
6.7.1 什么是故障复盘 215
6.7.2 为什么要做故障复盘 215
6.7.3 如何做故障复盘 216
6.7.4 故障复盘模板 219
6.8 监控与告警 221
6.8.1 监控的重要性 221
6.8.2 监控哪些内容 221
6.8.3 告警 224
6.8.4 监控的架构 224
6.9 应急处置 226

第 3 篇 管理探秘

第 7 章 为成为技术主管做准备 232
7.1 构建自我阶段性目标 232
7.2 体验自己的目标身份 234
7.3 勇于抓住机会 235
第 8 章 从自我管理转为管理团队 236
8.1 什么是领导力 236
8.2 如何构建领导力 237
8.2.1 维度 1：提供清晰的领导力风格，并以信任感作为基石 238
8.2.2 维度 2：了解业务，并带领团队达到高绩效 240
8.2.3 维度 3：发展自己和团队成员 240
8.2.4 维度 4：塑造未来 241
8.3 让自己成为 T 型人才 242

8.4 高效时间管理 243
8.4.1 确定在做的事情符合自己的目标 244
8.4.2 随时应用 80/20 法则 246
8.4.3 创造大块时间 247
8.5 遇到“不服管”的员工怎么办 249
8.5.1 每个人都是不错的 250
8.5.2 建立亲和与信任感 251
8.5.3 从绩效评估到建设性反馈 255
8.6 如何处理冲突 261
8.6.1 了解产生冲突的原因 261
8.6.2 正确看待冲突 262
8.6.3 处理冲突 263
8.7 引导员工主动工作 264
8.8 如何从带 10 个人到带 100 个人 266
8.8.1 在管理上面对的挑战 267
8.8.2 从管理 1.0 到管理 2.0 269
8.9 如何对上管理 276
8.9.1 做好对上的预期管理 277
8.9.2 及时汇报 277

1

第 1 篇

技术精进

第 1 章

程序员技能与成长

1997 年，笔者高中毕业，在懵懂状态下报考了计算机软件专业。为什么报考这个专业？其实第一个原因是感觉可以挣钱，不用再干农活；第二个原因是听说 21 世纪的人才要掌握三种必备的傍身技能：英语、计算机和开车。

时至今日，所言非虚。PC 早已广泛普及，智能手机、iPad 和智能终端已经“飞入寻常百姓家”。计算机课程教育虽然与时俱进，但离实际的企业应用还有很大的距离，仍有大量的内容需要在企业的项目、产品中进行实践，比如代码规范、软件测试、软件调试方法等。本书大部分作者从业十多年，经历了从程序员到架构师，再到 TL、公司总监的转变，相信本书的内容会让读者有所受益。

1.1 如何学习新的编程语言

一名软件工程师的最大挑战就是使自己的技术栈跟得上技术的发展，而在这个技术飞速发展的时代，保证自己不被淘汰的唯一方法就是不断学习。

那么，程序员需要掌握多门编程语言吗？很多初学者都被这个问题所困扰。Google 研究总监 Peter Norvig 曾就这个问题给出自己的观点：一名优秀的程序员至少应该掌握 6 门编程语言，其中包括支持类抽象的编程语言如 Java 或 C++，支持函数抽象的编程语言如 Lisp 或 ML，支持语义抽象的编程语言如 Lisp，支持声明规范的编程语言如 Prolog 或 C++模板，支持协程的编程语言如 Icon 或 Scheme，以及支持并发的编程语言如 Sisal。

一名画家若擅长使用多种类型的画笔，就可以创作出多种类型的艺术画作；一名程序员若掌握多种类型的编程语言，在解决问题时就可以有多种选择。

1.1.1　重点学什么

《计算机程序的构造和解释》的作者曾经表达这样的观点：在学习一门新的编程语言时，应该关注这门语言的基本表达形式（Primitive Elements）、组合的方法（Means of Combination）及抽象的方法（Means of Abstraction）这三个特性。

如果展开以上三个特性的话，就几乎包含了学习一门编程语言所需要关注的所有重要知识。

◎ 基础知识：基本语法、关键字、变量与常量、数据类型、运算符、流程控制、异常处理、文件处理、编程思想（面向对象、面向过程、函数式编程）、多线程支持等。
◎ 应用知识：网络请求、数据处理、内置函数、对日志和调试的支持、对单元测试的支持、序列化与反序列化等。
◎ 高级知识：开源类库、开源框架、底层原理等。

1.1.2　学习方法

学习编程需要长期坚持，不要迷信三五天就能让人学会的教程，你可能三五天掌握了一些语法，却难以完成复杂一点的编程，也没有和同行交流的经验。

1. 选择适合自己的编程语言

如果想学习一门新的编程语言，又不知道学习哪一门的话，可以参考 TIOBE 编程语言排行榜，其榜单每个月都会更新，可以反映某编程语言的热门程度。

在学习编程语言之前，需要先简单了解其主要特性及可以解决的问题，即选择适合自己的编程语言，带着目的去学习。

如表 1.1 所示为一张关于编程语言分类的表格，读者在选择编程语言进行学习时，可以参考这张表，选择最适合自己的编程语言。比如，你擅长 Java，希望学习另一门编程语言来提升自己的竞争力，就可以先选择和 Java 相似的编程语言，例如和 Java 同为强类型的、解释型的编程语言，再根据 TIOBE 编程语言排行榜就知道 Python 值得学习。

表 1.1

分　类	编程语言
动态类型语言	Python、Ruby、Erlang、JS、Swift、PHP、Perl
静态类型语言	C、C++、Java、Object-C
强类型语言	Java、C#、Python、Object-C、Ruby
弱类型语言	JS、PHP、C、C++
编译型语言	C、C++、Pascal、Object-C、Swift
解释型语言	Python、JS、Erlang、PHP、Perl、Ruby、Java
静态语言	C、C++、Java
动态语言	C#、Python、Object-C、PHP、Erlang、JS

当然，选择编程语言的标准不尽相同，比如想从事手机 App 开发，就需要学习和移动端开发有关的编程语言，比如 Object-C、Java 和 Swift；如果对区块链技术感兴趣，想要从事区块链开发，那么 Go、Python、Solidity、C++等会是不错的选择。

2. 选择好的学习方式

在学习新的编程语言时，笔者认为读书、看视频和参加培训都是不错的学习方式。当然，不同的学习方式适合不同的人。

很多人通过阅读书籍进行学习，因为他们觉得书籍上的内容相对完善且成体系，并认为通过视频和课程学习会比较慢。关于编程的书，大概有入门类、工具类、实战类、进阶类、原理类等，可以根据自己的知识程度进行选择，切勿盲目选择。

还有些人认为在阅读书籍的过程中会遇到很多没见过的名词、定义等，容易阻碍学习的进度。他们愿意选择偏重实践的内容，倾向于课程、视频等方式，因为可以进行现场敲代码、排查问题等。

各种学习方式并无好坏之分，适合自己才最重要。如果能够将多种方式相结合，通过书籍完善自己的知识体系，并提升理论知识，再通过视频及课程增加自己的实战经验，就再好不过了。

另外，在学习新语言时，翻阅官方文档和源码也是必不可少的，当然，这比较适合在学习的中后期进行。切勿遗漏这个步骤，这是了解并掌握一门语言的至关重要的步骤。

3. 勤加练习

很多开发者容易陷入误区，只注重理论知识的学习，不注重实战，在回答别人问题的时候头头是道，一旦动手实践却不知所措。所以，学习一门编程语言，是绝对离不开动手实践的。我们要把从书本中学习到的理论知识和实际应用结合起来，由浅入深地学习，最终达到熟能生巧的目的。孔子说“学而不思则罔，思而不学则殆”，在学习编程语言的过程中，学和思固然重要，勤加练习却也是必不可少的。

在学习编程语言的过程中进行练习，可以增加自己对理论知识的理解，增强自己的记忆。我们都知道，Java 中的 int 是有范围的，书本上说如果超过范围就会溢出，那么这个范围到底是多少，溢出之后的表现是什么呢？只有真正地敲一遍代码，真正地练习一下才会有深刻的体会，才能在日后的工作中避免发生类似的错误。

另外，在练习的过程中难免会遇到各种各样的问题。比如，Java 初学者在安装 JDK 和配置环境变量时可能会遇到很多问题，其想办法解决问题的过程非常可贵，因为在日后的工作中能够自主解决各类问题，是一名优秀程序员的必备技能。在很多时候，初级程序员和高级程序员之间最突出的区别其实就是解决问题的能力。通过实践，我们也可以锻炼自己在这方面的能力。所以，在实践的过程中遇到任何问题都不要退缩和逃避，要勇敢地面对并解决问题。

4. 带着问题学习

学习要由目标驱动，在目标驱动起作用后，我们还可以采用问题驱动方式进行学习，即在学习过程中多问问题。

问问题可以采用六何法。六何法，又叫作 6W 或 5W1H 分析法，即 What、Who、When、Where、Why 及 How，需要我们在学习的过程中多思考、多问问题。

举个简单的例子，在学习设计模式中的单例模式时，可以用六何法多提几个问题，例如：

◎ 什么是单例模式？
◎ 什么时候使用单例模式？
◎ 怎么实现单例模式？
◎ 哪种单例实现方式最好？
◎ 在单例模式中如何保证线程安全？

在学习的过程中，如果没被问题驱动，你学到的就可能只是一个技术概念和用法。有了六合法的问题驱动，你学到的就会从一个点横向扩展成一条线，如果将线上的每个点都逐渐深入，就会扩展成一个面。比如，我们用六合法来学习单例时，就可能以单例模式为中心点扩展到线程安全、锁、序列化、枚举、类加载机制等知识。

5. 教是最好的学

通过写博客来学习也是非常棒的一种学习方式，这对于新技术的学习十分有效，还可以通过技术分享、线下会议及线上教学等方式将自己学到的知识分享给他人，这就是教学学习法。

教学学习法有如下好处。

◎ 迫使自己更深入地了解更多的知识。
◎ 在教学的过程中会加入自己的理解。
◎ 可以回头翻看教学的内容。
◎ 可以加深记忆。
◎ 可以和别人深入探讨。

1.2 代码规范与单元测试

2017 年，阿里巴巴发布编码规范，这是开源界的一件大事，也在知乎等平台上引发了广泛的讨论，其中有个别回复纠结于具体细节的商榷和建议，但大部分认同该规范的指导意义。本节拟从代码规范、单元测试、代码审查及审查清单谈谈笔者的一些体会。

1.2.1 编码规范

不以规矩，不能成方圆。为什么要有规范（规约）想必不用多说了。本书重在讲述程序员如何具备大局观，具备大局观所要涵盖的知识宽度和视野，因此对于具体编码规范的逐条解析不作为重点。

Google Java Style Guide 包含的内容有源文件基础、源文件结构、格式、命名、编程实践和 Javadoc，可以作为一个团队必须遵守的共识。一套好的规范应该搭配好的审查清单（CheckList）。《Java 开发手册 1.5.0》就是一个不错的融合规约和审查清单的案例，其

中的单元测试一章有一条规范检查项，引用如下：

【强制】单元测试应该是全自动执行的，并且是非交互式的。测试框架通常是定期执行的，执行过程必须完全自动化才有意义。对输出结果需要人工检查的测试不是好的单元测试。在单元测试中不准使用 System.out 进行人肉验证，必须使用 Assert 进行验证。

该项可以作为一条独立的审查清单纳入 CheckStyle 或者 PMD 这样的工具来扫描静态代码。对于单元测试应该如何编写，会在下面的单元测试小节展开讨论。

1.2.2　单元测试

单元测试（Unit Testing，UT）说起来简单，在实际操作过程中却要注意它不是为了测试而测试的。如下所示是一段对 Java 中的字符串进行右对齐操作的代码：

```
/**
     * 扩展并右对齐字符串，用指定的字符串填充目标字符串的左边
     * @param str
     * @param size
     * @param padStr
     * @return str
     */
    public static String alignRight(String str, int size, String padStr) {
        if (str == null) {
            return null;
        }

        if ((padStr == null) || (padStr.length() == 0)) {
            padStr = " ";
        }

        int padLen = padStr.length();
        int strLen = str.length();
        int pads = size - strLen;
        if (pads <= 0) {
            return str;
        }

        if (pads == padLen) {
            return padStr.concat(str);
        } else if (pads < padLen) {
```

```
            return padStr.substring(0, pads).concat(str);
        } else {
            char[] padding = new char[pads];
            char[] padChars = padStr.toCharArray();

            for (int i = 0; i < pads; i++) {
                padding[i] = padChars[i % padLen];
            }

            return new String(padding).concat(str);
         }
    }
```

测试代码如下：

```
@Test
 public void testAlignRight ()
    {

      assertEquals(StringUtil.alignRight(null, 3, "a") , null);
      assertEquals(StringUtil.alignRight("", 3, "w") , "www");
      assertEquals(StringUtil.alignRight("bat", 3, "yw") ,"bat");
      assertEquals(StringUtil.alignRight("bat", 5, "yw") , "ywbat");
      assertEquals(StringUtil.alignRight("bat", 1, "yw") , "bat");
      assertEquals(StringUtil.alignRight("bat", -1, "yw") , "bat");

    }
```

如何采用测试驱动设计（Test-Driven Development，TDD）方式写这段业务代码呢？这里先看一下要满足的需求：按照指定的目标长度扩展并右对齐字符串，用指定的字符串填充目标字符串的左边。

所以，这个需求对应的测试用例如下。

◎ 如果源字符串为 null，则无论指定目标长度为多少，结果都为 null。
◎ 如果源字符串为""(空串),指定目标长度为 3、填充字符串为"w",则结果为"www"。
◎ 如果源字符串为"bat"，指定目标长度为 3、填充字符串为"yw"，则结果为"bat"。
◎ 如果源字符串为"bat"，指定目标长度为 5、填充字符串为"yw"，则结果为"ywbat"。
◎ 如果源字符串为"bat"，指定目标长度为 1、填充字符串为"yw"，则结果为"bat"。
◎ 如果源字符串为"bat"，指定目标长度为-1、填充字符串为"yw"，则结果为"bat"。

1.2.3　测试驱动设计

笔者在 2010 年做了一件现在看起来不靠谱的事情，就是把一个模块的代码测试覆盖率做到了 80%，在笔者刚接手时其覆盖率是 30%。为此，笔者让一位研发人员补测试补了一周，这可以算作为了覆盖率而做，但其效果如何，我们不敢抱太大希望，比如在下一次重构的时候，这些测试代码是否值得信赖。

有位咨询师提到，测试覆盖率低的病灶常常如下。

◎ 团队成员没有写测试的习惯，没有意识到写测试的重要性，不想写。
◎ 代码难于测试，不会写。
◎ 赶进度，没有时间写。

相对于提升测试的覆盖率，解决这些问题要复杂、棘手得多。笔者不确定上述问题在特定环境下的解决方法是什么，但很确定补测试不是良方，它往往会催生出没有 Assert 的畸形测试及大量针对 getter、setter 的无用测试。

这件事情给笔者一个教训，就是单元测试一定不能事后补。我们团队在项目实际操作过程中没有严格遵循测试驱动设计的步骤，但是遵循了同步编写开发代码和测试代码的思路。

测试驱动设计的基本思想就是在开发功能代码之前先编写测试代码，也就是说在明确要开发的需求之后，首先思考如何分析这个需求，并完成测试代码的编写，然后编写相关代码来满足这些测试用例，最后循环添加其他测试用例，直到该需求对应的测试用例都测试通过。

测试驱动设计有 3 个原则，如下所述。

◎ 原则 1：无测试，不代码。
◎ 原则 2：单元测试不在多，能够识别出问题即可。
◎ 原则 3：代码不在多，让当前单元测试全部通过即可。

下面看看具体的操作步骤，以 1.2.2 节的需求为例。

第 1 步：金丝雀测试

“金丝雀测试”的概念来自早期的煤炭矿井行业：金丝雀对有毒气体比较敏感，在 19 世纪左右，英国的矿井工人在下矿井时常常会带一只金丝雀，如果矿井内的有毒气体超标，金丝雀就会立刻死亡，这会救矿井工人一命。

同样，在测试驱动设计实践中，在开发具体的测试用例之前，也需要先写一个 dummy 的测试用例，确保整个编译、运行和 JUnit 环境是正常运行的。

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
/**
 * Created by brliu on 2018/5/7.
 */
public class StringUtilTest {
    @Test
    public void testAlignRight() throws Exception {
        assertTrue(false);
    }
}
```

之后运行这个测试用例，结果符合预期，说明整个编译、执行和 JUnit 环境是好的。

第 2 步：编写第 1 个最简单的单元测试

对应的需求为：如果源字符串为 null，则无论指定的目标长度为多少，结果都为 null。编写测试代码如下：

```
import org.junit.Ignore;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
/**
 * Created by brliu on 2018/5/7.
 */
public class StringUtilTest {
    @Ignore
    public void testAlignRight() throws Exception {
        assertTrue(false);
    }

    @Test
    public void testAlignRight_givenNull_returnsNull() throws Exception {
        assertEquals(MyStringUtil.alignRight(null, 3, "a"), null);
    }
}
```

好，在第 1 个单元测试运行时出现红色，说明编译出现了问题。开始编写如下代码：

```
    /**
     * Created by brliu on 2018/5/10.
     */
    public class MyStringUtil {
        public static String alignRight(String src, int outputLength, String
fillingString) {
            return null;
        }
    }
```

这样，第 1 个单元测试在运行时就是绿色的了。也许有人会问：“这段代码有用吗？”然而，从另一方面来说，这不就是实现当前需求的最简洁和最高效的实现吗？

第 3 步：编写第 2 个单元测试

对应的需求为：如果源字符串为""（空串），指定目标长度为 2、填充字符串为"w"，则结果为"ww"。编写单元测试，代码如下：

```
    @Test
    public void
testAlignRight_givenEmptyString_andFillingString_returnFillingString() throws
Exception {
        assertEquals(MyStringUtil.alignRight("", 2, "w"), "ww");
    }
```

测试时再次出现红色，将代码修改如下：

```
    public class MyStringUtil {
        public static String alignRight(String src, int outputLength, String
fillingString) {
            if (src == null) {
                return null;
            }

            String alignedStr = "";
            if (src.length() == 0 && fillingString.length() == 1) {
                StringBuilder strBuilder = new StringBuilder();

                for (int i = 0; i < outputLength; i++) {
                    strBuilder.insert(0, fillingString);
                }
```

```
            alignedStr = strBuilder.toString();
        }

        return alignedStr;
    }
}
```

这段代码非常具体（Specific），没有通用性（Generic），但是不妨碍非常高效地满足了当前的需求。

第 4 步：继续加需求

对应的需求为：如果源字符串为""（空串），指定目标长度为 3、填充字符串为"w"，则结果为"ww"。

这个需求其实是和上一个需求等价的，所以可以把这个单元测试和上一个单元测试合并：

```
    @Test
    public void
testAlignRight_givenEmptyString_andFillingString_returnFillingString() throws
Exception {
        assertEquals(MyStringUtil.alignRight("", 2, "w"), "ww");
        assertEquals(MyStringUtil.alignRight("", 3, "w"), "www");
    }
```

再运行一次，结果怎样呢？运行结果正确！

第 5 步：接着加需求

对应的需求为：如果源字符串为"abc"，指定目标长度为 3、填充字符串为"w"，则结果为"abc"。

增加测试用例，代码如下：

```
    @Test
    public void
testAlignRight givenSrcString andFillingString returnSrcStringFirst() throws
Exception {
        assertEquals(MyStringUtil.alignRight("abc", 3, "w"), "abc");
    }
```

运行一次单元测试，发现运行失败。看来我们要继续写代码了，再加入一个 if 块：

```
    public class MyStringUtil {
        public static String alignRight(String src, int outputLength, String
fillingString) {
            if (src == null) {
                return null;
            }

            String alignedStr = "";
            if (src.length() == 0 && fillingString.length() == 1) {
                StringBuilder strBuilder = new StringBuilder();

                for (int i = 0; i < outputLength; i++) {
                    strBuilder.insert(0, fillingString);
                }

                alignedStr = strBuilder.toString();
            }

            if (src.length() >= outputLength) {
                StringBuilder stringBuilder = new StringBuilder();

                stringBuilder.insert(0, src);
                alignedStr = stringBuilder.toString();
            }

            return alignedStr;
        }
    }
```

第 6 步：再次加需求

对应的需求为：如果源字符串为"abc"，指定目标长度为 5、填充字符串为"wxy"，则结果为"wxabc"。相关测试代码如下：

```
    @Test
    public void
testAlignRight_givenSrcString_andFillingString_returnSrcStringFillingStrCombined
() throws Exception {
        assertEquals(MyStringUtil.alignRight("abc", 5, "wxy"), "wxabc");
    }
```

实现代码如下：

```
    public class MyStringUtil {
        public static String alignRight(String src, int outputLength, String
fillingString) {
            if (src == null) {
                return null;
            }

            String alignedStr = "";
            if (src.length() == 0 && fillingString.length() == 1) {
                StringBuilder strBuilder = new StringBuilder();

                for (int i = 0; i < outputLength; i++) {
                    strBuilder.insert(0, fillingString);
                }

                alignedStr = strBuilder.toString();
            } else if (src.length() >= outputLength) {
                StringBuilder stringBuilder = new StringBuilder();

                stringBuilder.insert(0, src);

                alignedStr = stringBuilder.toString();
            } else if (src.length() < outputLength) {
                StringBuilder stringBuilder = new StringBuilder();

                stringBuilder.insert(0, src);

                stringBuilder.insert(0, fillingString.substring(0, outputLength -
src.length()));

                alignedStr = stringBuilder.toString();
            }

            return alignedStr;
        }
    }
```

第 7 步：重构，是为了更好地前行

到目前为止，我们的代码已经“生长”到 30 行了，现在选择重构当前代码，主要关注两方面：让代码更整洁；让应用“从特殊到一般”来泛化代码，使“算法”更清晰。

首先，为了提取“pattern”，我们发现第 1 个 if 块和第 3 个 if 块有些类似。为了让它们呈现一样的“pattern”，我们在第 1 个 if 块中加入一条 dummy 语句：

```
    public class MyStringUtil {
        public static String alignRight(String src, int outputLength, String
fillingString) {
            if (src == null) {
                return null;
            }

            String alignedStr = "";

            if (src.length() == 0 && fillingString.length() == 1) {
                StringBuilder strBuilder = new StringBuilder();

                // 加入的 dummy 语句
                strBuilder.insert(0, src);

                for (int i = 0; i < outputLength; i++) {
                    strBuilder.insert(0, fillingString);
                }

                alignedStr = strBuilder.toString();
            } else if (src.length() >= outputLength) {
                StringBuilder stringBuilder = new StringBuilder();

                stringBuilder.insert(0, src);

                alignedStr = stringBuilder.toString();
            } else if (src.length() < outputLength) {
```

然后，为了发掘 pattern，对下面的代码块进行重构：

```
            if (src.length() == 0 && fillingString.length() == 1) {
                StringBuilder strBuilder = new StringBuilder();

                // 加入的 dummy 语句
```

```
            strBuilder.insert(0, src);

            for (int i = 0; i < outputLength; i++) {
                strBuilder.insert(0, fillingString);
            }

            alignedStr = strBuilder.toString();
        }
```

这段代码的逻辑是，当输入的源字符串长度为零，而填充字符的长度为 1 时，重复利用填充字符进行填充。我们可以重构“重复利用填充字符进行填充”这段代码：

```
    // 将重复利用填充字符进行填充的准备工作提取出来
    if (fillingString.length() == 1) {
        StringBuilder strBuilder = new StringBuilder();

        for (int i = 0; i < outputLength; i++) {
            strBuilder.insert(0, fillingString);
        }

        processedFillingStr = strBuilder.toString();
    } else {
        processedFillingStr = fillingString;
    }
```

这样的话，就可以把这个 if 块重构为和其他 if 块相似的 pattern：

```
    if (src.length() == 0 && fillingString.length() == 1) {
        //现在这个 if 块就和下面第 3 个 if 块有一样的 pattern，可以进行合并了
        StringBuilder strBuilder = new StringBuilder();

        strBuilder.insert(0, src);

        strBuilder.insert(0, processedFillingStr.substring(0, outputLength -
src.length()));

        alignedStr = strBuilder.toString();
    }
```

这样一来，就可以通过合并第 1 个 if 块和第 3 个 if 块进行重构：

```
    if (src.length() == 0 && fillingString.length() == 1) {
        StringBuilder strBuilder = new StringBuilder();
```

```
        strBuilder.insert(0, src);
        strBuilder.insert(0, processedFillingStr.substring(0, outputLength -
src.length()));

        alignedStr = strBuilder.toString();
    } else if (src.length() >= outputLength) {
        StringBuilder stringBuilder = new StringBuilder();

        stringBuilder.insert(0, src);

        alignedStr = stringBuilder.toString();
    } else if (src.length() < outputLength) {
        StringBuilder stringBuilder = new StringBuilder();

        stringBuilder.insert(0, src);
        stringBuilder.insert(0, processedFillingStr.substring(0, outputLength -
src.length()));

        alignedStr = stringBuilder.toString();
    }
```

重构的结果如下：

```
    if (src.length() >= outputLength) {
        StringBuilder stringBuilder = new StringBuilder();

        stringBuilder.insert(0, src);

        alignedStr = stringBuilder.toString();
    } else {
        StringBuilder strBuilder = new StringBuilder();

        strBuilder.insert(0, src);
        strBuilder.insert(0, processedFillingStr.substring(0, outputLength -
src.length()));

        alignedStr = strBuilder.toString();
    }
```

第 8 步：继续加入需求

对应的需求为：如果源字符串为"abc"，指定目标长度为 1、填充字符串为"wxy"，则结果为"abc"。相关测试代码如下：

```
    @Test
    public void
testAlignRight_givenSrcString_andOutputLengthLessThanSrcString_returnSrcString()
throws Exception {
        assertEquals(MyStringUtil.alignRight("abc", 1, "wxy"), "abc");
    }
```

运行单元测试，结果居然是通过！

第 9 步：继续加入需求

对应的需求为：如果源字符串为"abc"，指定目标长度为-1、填充字符串为"wxy"，则结果为"abc"，也就是说不进行处理。相关测试代码如下：

```
    @Test
    public void
testAlignRight_givenSrcString_andOutputLengthLessThanSrcString_returnSrcString()
throws Exception {
        assertEquals(MyStringUtil.alignRight("abc", 1, "wxy"), "abc");
        assertEquals(MyStringUtil.alignRight("abc", -1, "wxy"), "abc");
    }
```

运行单元测试，结果仍然是通过。

也就是说，在前面的重构中，在使用了“从特殊到一般”对代码进行泛化重构之后，代码的使用范围更广了，或者说隐藏在背后的算法显现了，这也说明测试驱动设计是可以演化出算法的。

1.3 使用静态代码分析工具

前面的小节介绍了一些编码规范及最佳实践，如果在编码阶段可以很好地遵循这些规范，并践行最佳实践的话，就可以大大提升代码的质量、提高可读性、降低维护成本等。在很多公司内部及开发团队中，为了保证大家都遵循这些规范，都会有静态代码分析（Static Code Analysis）流程来保证在上线前检测出代码缺陷。

本章主要介绍静态代码分析技术相关的方法和工具，在掌握了这些方法和工具后，就可以很好地进行静态代码分析了。

1.3.1　什么是静态代码分析

静态代码分析是指在不运行计算机程序的条件下进行程序分析的方法。静态代码分析仅通过分析代码的词法、语法、语义、控制流等来检查代码的正确性，帮助软件开发人员、质量保证人员查找代码中的结构性错误、不符合代码规范的地方、安全漏洞等，从而保证软件的整体质量。

在进行静态代码分析时不需要运行被测试的代码，所以一般在编码阶段就可以发现问题。在整个软件开发生命周期中，大概有 30%～70%的代码逻辑设计和编码缺陷是可以通过静态代码分析发现和修复的。

由于静态代码分析需要分析者有很多的经验积累，并且极其耗费时间，因此对于很多软件开发团队来说，使用静态代码分析工具自动化执行代码审查和分析，能够极大提高软件的可靠性并节省软件开发和测试的成本。

目前有很多静态代码扫描工具可以作为 IDE 的插件，在编写代码阶段进行实时静态扫描，实时提示代码缺陷，可以帮我们大大节省做代码审查和修复的时间。

1.3.2　静态代码分析工具

在代码构建过程中使用静态分析工具有着诸多好处，所以在市面上有很多成熟的静态代码分析工具。本节重点讲解静态代码分析工具 Checkstyle、FindBugs 及 P3C 的使用方法。

1. Checkstyle

Checkstyle 是 SourceForge 的开源项目，是一个帮助 Java 开发人员遵循某些编码规范的工具。它能对代码的编码格式、命名方式、Javadoc、类设计等进行自动化的代码规范和风格检查，从而有效约束开发人员更好地遵循代码编写规范。

Checkstyle 目前几乎支持所有主流的 IDE，在 IntelliJ IDEA 和 Eclipse 上都有 Checkstyle 的插件。

Checkstyle 支持用户根据需求自定义代码审查规范，用户可以在已有检查规范如命名约定、Javadoc、块、类设计等基础上添加或删除自定义检查规范。Checkstyle 插件还可以直接导入代码审查配置文件，例如可以直接使用 Checkstyle 配合 Google Java Style Guide 配置文件对 Java 代码进行静态扫描。

在 IntelliJ IDEA 设置菜单中选择“Plugins”菜单项，单击中下方的“Browse repositories...”按钮，即可通过网络搜索并安装 Checkstyle 插件，如图 1.1 所示。

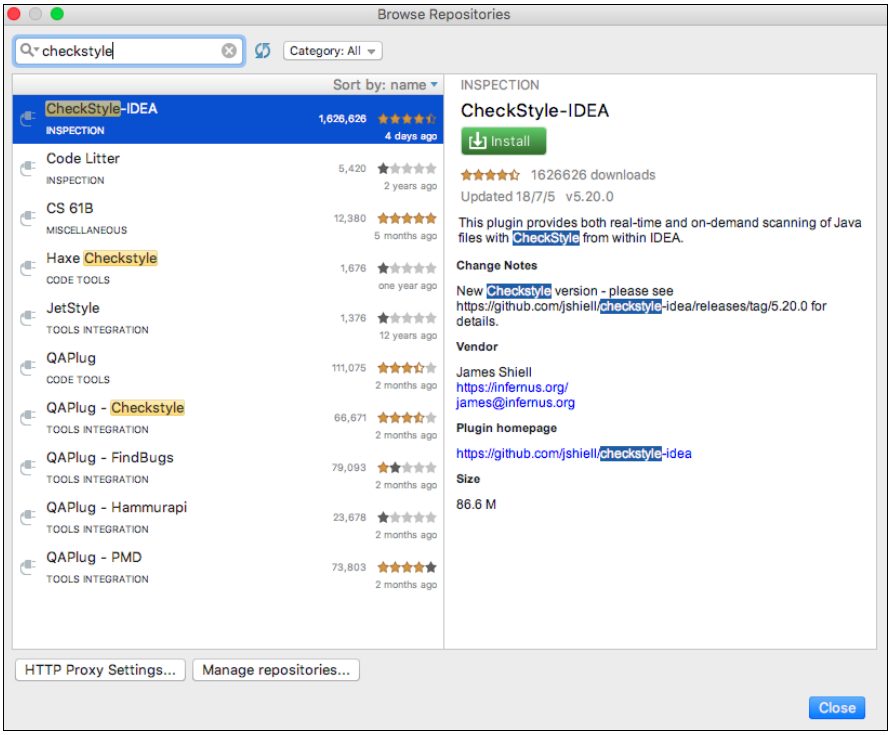


图 1.1

在安装好 Checkstyle 后，便可以在 IntelliJ IDEA 的“Other Setting...”选项中找到 Checkstyle 的设置入口，进入修改 Checkstyle 的设置页面。下面通过 Checkstyle 使用 Google Java Style Guide 进行代码扫描，如图 1.2 所示。

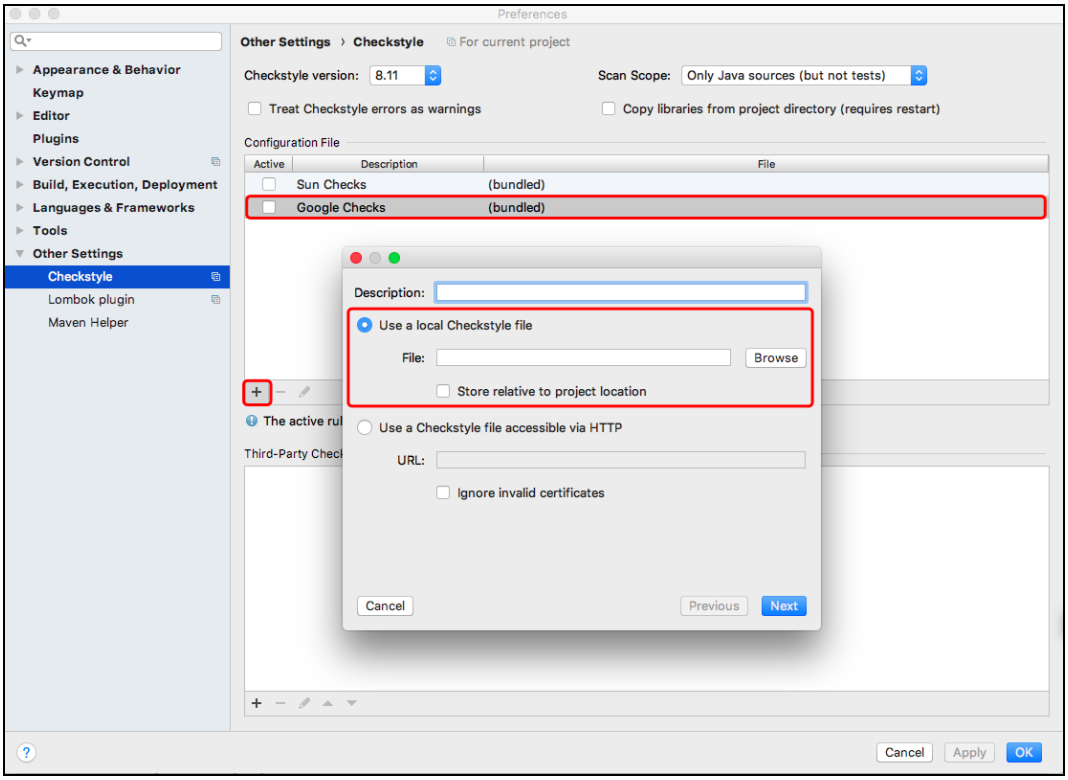


图 1.2

在配置好 Checkstyle 后就可以扫描代码了。Checkstyle 有一个单独的视图，其中包含代码扫描、规则选择、代码缺陷列表展示等功能，如图 1.3 所示。

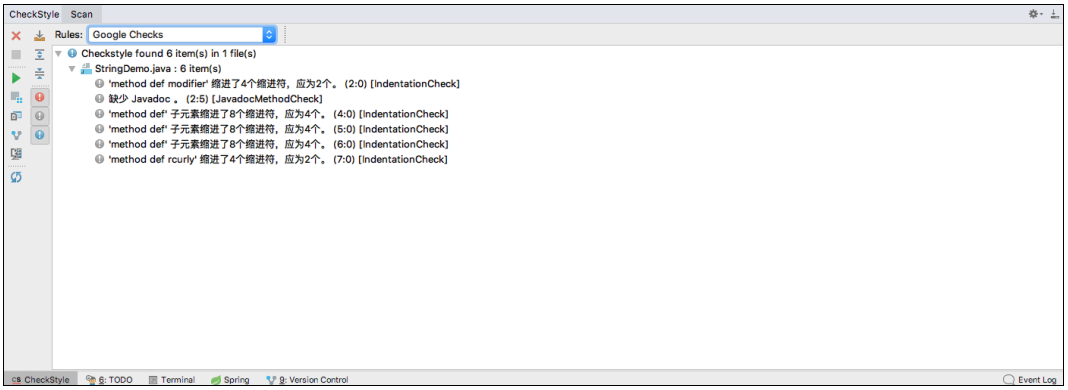


图 1.3

Checkstyle 会根据配置的规则扫描代码，并列举扫描到的代码缺陷，开发者可以通过如图 1.3 所示的视图查看有缺陷的代码，并选择性地进行修复。

2. FindBugs

FindBugs 是马里兰大学开源的一款主要针对 Java 语言的静态代码分析工具，通过检查类文件或 JAR 文件，将字节码与预先定义好的缺陷模式进行对比来发现代码缺陷，以达到静态代码分析的目的。和 Checkstyle 一样，FindBugs 既可以提供可视化 UI 界面，也可以作为 Eclipse 插件使用。

在 IDE 的 FindBugs 视图中需要先选择待扫描的.class 文件，如果已经有这些.class 对应的源文件，就可以选择这些.java 文件，这样便可以从稍后得出的报告中便捷地定位代码缺陷。还可以选择工程使用的 library，这样可以帮助 FindBugs 做一些高阶检查，以发现更深层的一些 Bug。

在选择以上各项后，便可以进行检测了，之后可以生成一份详细的报告。我们根据这份报告，就可以发现在许多代码中潜在的 Bug。FindBugs 的扫描结果有一个单独的视图页面，它会根据代码缺陷的性质，把扫描出来的代码缺陷大致分为以下几类。

◎ Dodgy code：糟糕的代码。
◎ Bad practice：不好的做法。
◎ Correctness：可能不正确。
◎ Experimental：实验。
◎ Internationalization：国际化。
◎ Malicious code vulnerility：恶意的代码漏洞。
◎ Multithreaded correctness：多线程问题。
◎ Performance：性能问题。

如图 1.4 所示是使用 FindBugs 对一段代码扫描后得到的代码缺陷列表，其中包含了前面提到的 Dodgy code、Bad partice、Correctness 及 Experimental 等 4 类缺陷。很多代码中的常见问题，例如引用了空指针（null pointer dereference）、多余的代码（useless code）等都可以被轻易检测出来。FindBugs 除了会提示代码有哪些问题，还会提供一些问题成因及修改意见等，可以很好地帮助开发人员在修改代码缺陷的同时提升自己的编码能力。

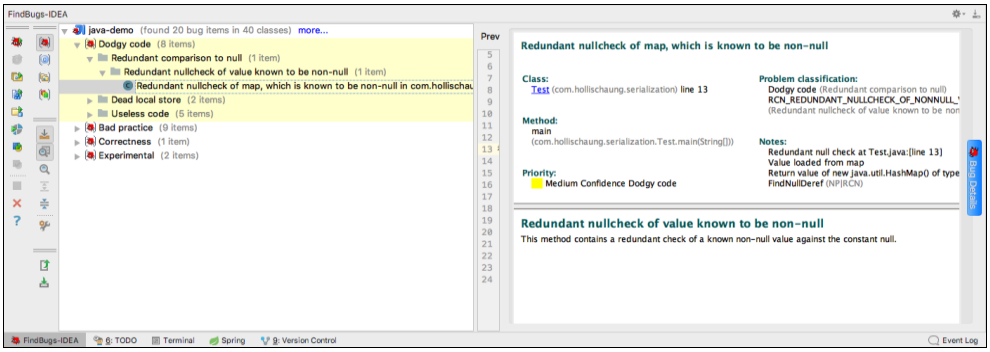


图 1.4

3. P3C

在 2017 年 10 月 14 日举办的杭州云栖大会上，Java 代码规约扫描插件全球首发仪式正式启动，《阿里巴巴 Java 开发规约》（后面简称“规范”）正式以插件形式公开走向业界，引领 Java 语言的规范之路。该插件是基于规范进行静态代码分析的工具，同样支持目前流行的多种 IDE，例如 IDEA、Eclipse 等。

该插件在扫描代码后，将不符合规约的代码按 Blocker、Critical、Major 三个等级显示在下方甚至 IDEA 上。该插件还基于 Inspection 机制提供了实时检测功能，在编写代码的同时能快速发现问题所在。而且，该插件能给出代码修改建议，还能对部分规则提供一键修改功能，方便开发者修改有问题的代码。

P3C 同样在 IDE 中提供了单独的视图，可以进行代码扫描及代码缺陷查看，如图 1.5 所示为通过 P3C 扫描后得到的代码缺陷列表及详情。

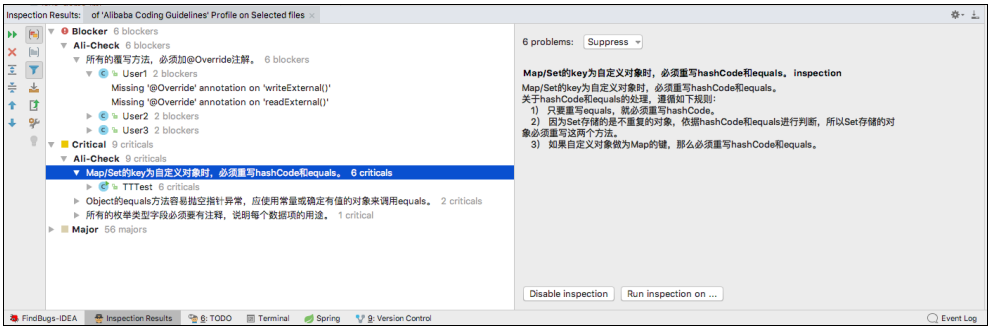


图 1.5

相对于 FindBugs 等其他工具，阿里巴巴推出的 P3C 插件对代码缺陷的描述及改进意见十分详细，是开发人员提升代码质量的必备工具。

1.4 代码审查

前面几节介绍了编码规范、单元测试、TDD 等，同时介绍了一种使用静态代码分析工具进行静态代码扫描，以帮助我们更好地发现代码缺陷的方法。对于在上线前发现代码缺陷，还有一种十分重要及奏效的方法，那就是代码审查（Code Review）。本节详细介绍什么是代码审查、代码审查的好处及如何做好代码审查。

1.4.1 什么是代码审查

代码审查，指对计算机源代码进行系统审查，以找出并修正在软件开发初期未发现的错误，提升软件的质量。代码审查可以帮助开发者了解自己的代码缺陷，从而提醒开发者避免类似问题的发生。相较于静态代码扫描，代码审查主要借助开发者之间的相互阅读及代码审查，来发现代码中的 Bug 等问题。

1.4.2 代码审查的好处

代码审查的重要性是被普遍认可的，因为通过代码审查可以提升代码的质量，找出潜在的 Bug，保证代码风格的统一，等等。但是，笔者认为，其最大好处往往被人忽略，即它是一个可以相互学习的好机会，比如审查者（Reviewer）可以在代码中发现他人的优秀编码习惯、对优秀设计模式的使用及使用了哪些好的 API 等，对了解整个系统和业务也有很好的帮助；另外，通过他人提出的一些改进意见，被审查者可以很好地学习到一些思考方式，在开发过程中有时“当局者迷”，审查者则可以从旁观者的角度重新审查代码及代码背后的设计思想。

所以，关于代码审查，笔者认为，其好处不仅仅是代码审查自身，还是在代码审查过程中产生的思考及思考之间的碰撞。

1.4.3　如何做代码审查

很多人都知道代码审查的重要性，也在项目中严格地进行代码审查，但是代码审查的效果并不明显。其实，这主要是因为进行代码审查的方法不对。接下来，笔者结合自己的经验介绍一下如何做好代码审查。

1. 要有审查清单

代码审查这件事其实是可以标准化和流程化的，其效果在一定程度上也是可以量化的。要想做好代码审查，尤其是对一个开发团队来说，有一份审查清单是非常有必要的，这样大家就都有了一个标准，可以在代码审查过程中按照审查清单逐一检查。使用审查清单可以帮助审查者快速找到问题，甚至开发者在开发阶段就可以按照审查清单进行代码自查。

网上有很多代码审查清单，但是不同的团队可能需要不同的代码审查清单，这需要根据团队的实际情况进行调整和完善。

一般来说，一份代码审查清单应该包括如下几个大的类目。

◎ 代码结构：是否包含超长代码，代码层次嵌套是否过深，函数是否入参过多，循环条件是否有跳出点，if 语句是否有对应的 else 语句，是否存在重复的代码，等等。

◎ 代码安全性：I/O 流是否正常关闭，资金计算是否使用了 Double 数据类型，是否有超大的临时对象，线程池大小是否合理，异常是否被忽略，是否有详细的日志记录，是否存在并发问题，参数是否做了必要的检查，远程服务的入参出参是否实现了 Serialization 并且自定义了 serialVersionUID，应用是否依赖了 SNAPSHOT 版本的类库，等等。

◎ 代码性能：是否有长 SQL 语句、SQL 语句是否用到索引，是否有成熟的类库可以替换自己实现的代码，是否可以考虑单例模式，是否可以考虑线程池，是否可以考虑 NIO，是否可以进行锁优化，等等。

◎ 代码注释：指类及方法是否有注释，注释是否可以表达其准确含义，在代码中是否存在 FIXME 及 TODO 等注释，注释是否包含边界值及对异常情况的说明，等等。

◎ 单元测试：代码是否有可测试性，新代码是否有单元测试，单元测试是否可以覆盖所有场景，等等。

◎ 代码优化：是否可以使用枚举代替自定义的常量，在代码中是否包含魔法值，是否可以使用 Optional 代替 NPE 的检查，是否可以使用 Stream 代替 for 循环，是否可以使用设计模式，等等。

◎ 其他：代码逻辑是否正确，是否实现了业务功能，代码是否有好的可读性及可测试性，等等。

2. 选择高效的工具

工欲善其事，必先利其器，选择一款高效的工具，可以大大提升代码审查的效率。这里推荐几款代码审查工具，如下所述。

1）Phabricator

Phabricator 最早是 Facebook 的内部审查工具，目前已经开源，可以在 Apache 许可证的第 2 版下作为自由软件分发。它是一套基于 Web 的软件开发协作工具，包括代码审查工具 Differential、资源库浏览器 Diffusion、变更监测工具 Herald、Bug 跟踪工具 Maniphest 和维基工具 Phriction。Phabricator 可与 Git、Mercurial、Subversion 集成使用，如图 1.6 所示。

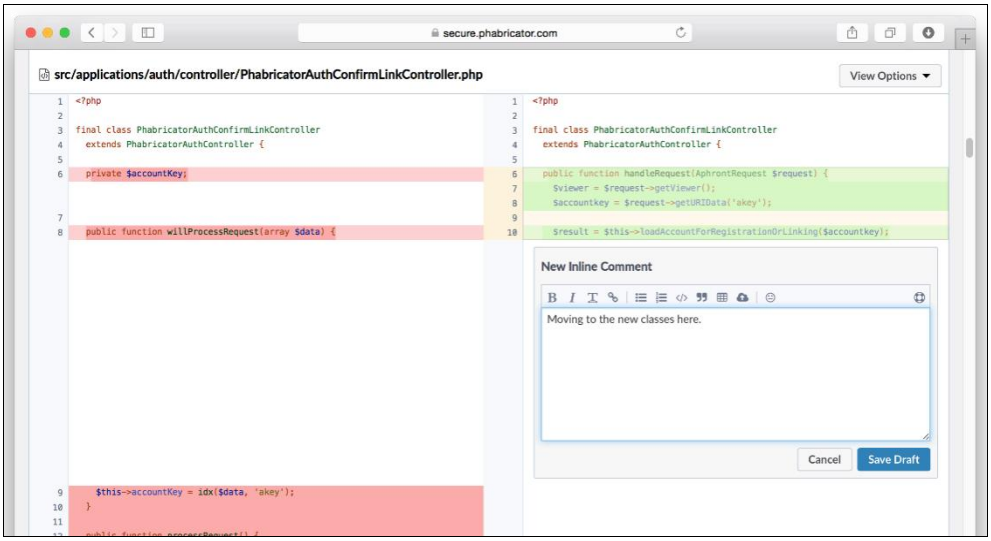


图 1.6

2）Gerrit

和 Phabricator 一样，Gerrit 也出身名门。Gerrit 是 Google 为 Android 系统研发量身定制的一套免费的开源代码审查系统，它与 Git 集成并建立在 Git 版本控制系统之上，允许在写完代码后将更改合并到 Git 存储库。它在传统的源码管理协作流程中强制引入了代码审查机制，通过人工代码审查和自动化代码验证过程，将不符合要求的代码屏蔽在代码库之外，确保对核心代码有多人校验、多人互备和自动化构建核验。

3）Code Striker

Code Striker 是基于 Perl 语言的工具，和其他工具一样，需要安装在服务器上。它支持对 CVS、Subversion、Clearcase、Perforce、Visual SourceSafe and Bugzilla 等工具的集成。

3. 重点关注的改动点

通过使用代码审查工具，我们可以很清楚地看出代码有哪些改动点（包括新增的代码），通过对照我们事先置顶的审查清单，重点关注改动点是一种比较高效的代码审查方式。原有的代码一般是经过代码审查的，但代码的变化往往会引入新的问题，所以，在审查代码的过程中应该重点关注代码的改动点。

但是，重点关注改动点并不是只关注改动点，其实，还有很多问题隐藏在非改动点处。这就需要负责代码审查的开发者通读改动点附近的全部代码，重点关注改动点，而不是只看改动点处的代码。通过通读全部代码，可以更好地发现问题，尤其是和业务逻辑相关的问题。

4. 代码审查应该是日常性的工作

代码审查应该是日常性的工作，而不是代码上线前的集中性工作。换句话说，代码审查应该是伴随着代码提交的，而不是伴随着代码发布的。最好的实践是在每次提交改动较大的代码后都找到对应的开发人员进行代码审查，这样可以更早地发现问题，也可以使其他人更早地了解这部分新代码。如果等到项目即将发布才集中进行代码审查，就会有很多问题，这时会有很多代码需要审查，导致审查人员过于疲劳，无法集中精力审核代码，并忽略一些问题。而且，在项目即将发布时进行代码审查，即使在代码审查中发现一些问题，也不便于修改。

5. 不要一次审查太多代码

2006 年 5 月，Smart Bear 针对 Cisco 进行了为期 10 个月的代码审查方面的研究，最终得到一份思科代码审查报告（*Code Review at Cisco Systems*，参见 http://support.smartbear.com/support/media/resources/cc/book/code-review-cisco-case-study.pdf）。在该报告中指出，做代码审查，每次审查的代码行数最好在 200 行以内，不超过 400 行，否则查找代码缺陷的效果就会大打折扣。

图 1.7 展示了一次审查的代码行数和可发现的代码缺陷之间的关系。我们发现，当代码行数超过 200 时，缺陷密度（每千行代码的缺陷数，为缺陷数量与代码行或功能点数量的比值，其测量单位是 defects/KLOC）就会急剧减小，超过 400 后，缺陷密度几乎为 0。

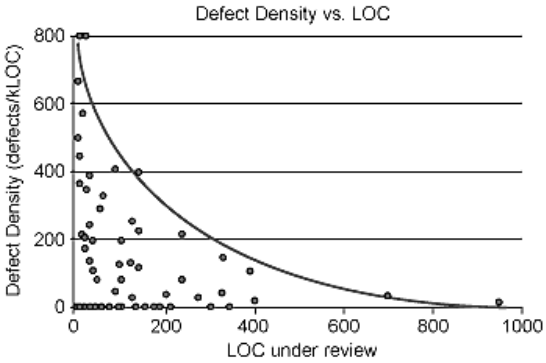


图 1.7

所以，每次代码审查的代码最好在 200 行左右，最多不要超过 400 行，这样更有利于代码审查者集中精力，最好每天都抽出固定的一段时间，来集中精力对少部分代码做代码审查。

6. 进行一次代码审查的时间不要太长

这里建议一次不要审查太多代码，进行一次代码审查的时间也不宜过长。研究表明，进行一次代码审查的时长应控制在 90 分钟内，这里建议控制在 60 分钟以内，如果审查时长超过 90 分钟，则代码审查的效率会大大降低。也就是说，虽然花费了更多的时间，但得到了更糟糕的结果。

在进行代码审查之前，应该规划好需要审查的代码长度及时长，尽量控制在 200 行代码、60 分钟左右，这样的效率是最高的。

1.5　清单和模板

人类的学习模式是向过去学习、向经验学习。我们“迷信”经验的力量，用已有的知识来解决新的问题，这也往往奏效。为了把这些经验传承下去，我们发明了清单和模板。我们可以将清单定义为 List，比如 CheckList。模板对于照猫画虎的知识传承尤其有用，在 IT 领域也有各种模板，比如产品需求文档、概要设计文档等。

1.5.1　清单实践

制定清单和使用清单，清单实践就这么简单吗？答案是否定的，可以先来读一个故事。

2005 年 8 月 29 日，卡特里娜飓风横扫国外某市，部分地区受灾严重，洪水肆虐，通信中断，消息无法送出。当人们最终将灾情反映到某个联邦官员时却被告知耐心等待。因为需要将相关信息逐级上报，所以传统的指令和控制系统很快被大量的信息和指令淹没。政府拒绝放弃传统的指令模式，灾情正在不断恶化，各级政府却在争论决策权的归属问题，满载饮用水和食物的卡车迟迟不能进入灾区。

我们可以从这个故事总结出下几点。

◎ 面对复杂的问题，做决策往往不容易。各级政府在争论决策权时，事态已经在恶化了。
◎ 要建立应急处理机制，简单、直接、有效。
◎ 要有应对各种问题的清单。

简单总结一下，在解决问题时要有如下几个步骤。

（1）出了什么问题，汇总问题和现象，试着探究原因。

（2）找到解决问题的流程，应该谁负责驱动，流程有哪些环节，有哪些可选方案。

（3）按步骤解决问题。

世界卫生组织的手术安全清单（Safe Surgery CheckList）由 19 个检查项目构成。其中，在实施麻醉前有如下 7 个检查项目。

◎ 患者本人或家属是否已经确认了患者的身份，并同意进行手术。
◎ 手术部位是否已经进行了标记。
◎ 是否给患者进行了血氧饱和度监测，仪器运转是否正常。

◎ 患者是否有既往过敏史。
◎ 是否存在气道困难和误吸的风险（这是实施全身麻醉最危险之处），以及所需设备和辅助人员是否已经就位。
◎ 是否存在失血量大于 500 毫升的风险，儿童为每千克 7 毫升。
◎ 必需的中心静脉置管、血袋和补液是否已经准备好。

由此可见，清单需要具备简洁、直接、易操作的特点。简洁是指不面面俱到，如果某个环节的检查项目过多，比如超过 20 项，则一不易操作，二容易出错。直接、易操作是指对清单的描述要清晰、无二义性、具体，对其效果应该能进行观测，并由此进行改进。

如表 1.2 所示为一份代码审查清单，这份清单仅总结了代码审查在安全维度方面的一些审查点。

表 1.2

分　　类	清单项目	安全策略
访问限制	最小化类和成员的可访问性	基础
	权限限制：程序应该运行在保证功能正常的最小权限模式下	基础
	限制包、类、接口、方法和域的可访问性	可访问性的扩展
	限制类和方法的可扩展性（使它为 final）	可访问性的扩展
资源管理	对于线程，线程资源必须通过线程池提供，不允许在应用中自行显式创建线程	受控管理
	在任何情况下都释放资源（流、连接等）	拒绝服务
	合理设置相关资源的配置参数	拒绝服务
线程安全	线程上下文变量的设置与清除必须配对	并发风险
	静态 Util 或单例必须是线程安全的，DateFormat 类是非线程安全的，在作为量使用时建议采用包装 DateUtil 工具类	并发风险
输入输出	对系统的输入必须检查是否有效和在允许的范围内	拒绝服务
	检验输入（有效的数据、大小、范围、边界情况等）	输入检验
	把从不可信对象那里得到的输出作为输入来检验	输入检验
	把从不可信对象那里得到的输出作为输入来对待	可变性
异常管理	从异常中清除敏感信息（暴露文件路径、系统内部相关配置）	私密信息
	不要把高度敏感的信息写到日志中	私密信息
	考虑将高度敏感的信息在使用后从内存中清除	私密信息

成也清单，败也清单。执行者在面对太多的清单内容时很容易体力疲劳、出问题，所以最好将流程和审查本身都强制到工具中。

“在提交代码时，强制要求有代码审查记录”就是一种强制流程。前文介绍过的静态扫描工具如 Checkstyle、FindBugs 都可以解决一些可以规则化、模板化的问题，比如代码基础规范大小写、缩进等。

要让清单成为一种习惯，清单就应该足够简单、清晰并且深入人心。

同时，笔者提倡把清单嵌入流程中，只有在流程中，才能受到关注和持续运营，从侧面治理的模式往往会名存实亡。笔者曾经在传统软件企业工作过几年，还记得每周或者每个月，SQA 和 SCM 人员都会拿着长长的 Excel 表让研发负责人或者 PM 打钩，表示“做了某项工作”。这种审查和度量模式的最大问题是 SCM 和 SQA 人员会对质量和流程遵循情况做展示并反馈给领导，但是研发工程师对“清单”并不感兴趣。后来笔者来到互联网方面的研发公司，SQA 的存在感几乎为零，但是我们在研发过程中形成了相关的“完成定义”，比如交付报告。研发人员必须在完成代码之后进行自测、静态代码评估和持续集成报告等，并在这些被判定为“完成”之后，才能进入下一个流程。

1.5.2　产品需求文档模板

下面给出产品需求文档模板，如表 1.3 所示。

表 1.3

1. 文档简介
产品需求说明书文档的简介应提供整个文档的概述，包括此产品需求说明书文档的目的、范围、定义、首字母缩写词、缩略语、参考资料和概述。
1.1　文档目的
阐明此产品需求说明书文档的目的，例如“本文档为小明 v1.0.0 的产品需求文档，主要作为确认需求及系统分析设计的依据”。
1.2　文档范围
简要说明此产品需求说明书文档的范围、相关产品，以及受到此文档影响的任何其他事物。
2. 用户角色描述
主要介绍产品的目标用户角色，包含前后端用户、各种平台、运营人员等，以及相关描述，如果可以的话，进行用户画像描述。

3. 产品概述

主要是高度概括产品的功能与介绍：

- 目标；
- 描述产品的目标；
- 总体流程；
- 描述产品的总体流程图；
- 功能摘要；
- 简要描述产品的功能点和每个功能点的优先级。

4. 产品特性

列出产品的特性。特性是为让用户获益而必须提供的高级系统功能，每项特性都是外部所需的服务，它通常需要一系列输入来实现预期的效果。本节为设计的系统功能性需求，一般以用例结合自然语言来表达。本节通常按特性来组织，但也可能会有其他适用的组织方式，例如按用户或子系统组织的方式。本节应包含所有的产品需求，其详细程度应能使架构设计人员和软件需求设计人员设计出可以满足这些需求的系统，不包括可选流程和异常流程，不对具体语义做约束。

4.1 功能模块

（1）产品概述

概述功能模块的产品特性及效果。

（2）产品结构（功能摘要）

概述功能模块的产品结构或包含组件。

（3）状态说明

列出产品的各种状态及状态转换图。

（4）特性说明

对特性的名称、功能点进行介绍，进行需求详述和一些补充说明。

4.2 其他产品需求

从业务视角提出各项可用性指标的大致需求。具体的技术指标会体现在产品的设计文档中（根据项目的实际情况进行增删）。

4.2.1 性能需求

如果产品对性能有特殊需求，则请详细描述，例如大致响应时间、最大并发数等。

4.2.2 监控需求

如果产品需要特殊的监控和统计，则请详细描述，例如 PV、点击率、登录数等。

4.2.3 兼容性需求

如果产品需要对兼容性提出特殊的需求，则请详细描述，例如 Web 应用兼容 IE8、Chrome，以及 App

兼容系统的什么版本，等等）。

5. 风险分析

风险的内容描述，说明风险产生的原因、可能造成的危害及相应出现的频率信息，还需要描述相关风险预防措施及风险出现后的应对措施信息。此处不包括任何系统技术实现层面的风险，例如系统的备份、监控、模块依赖等。

6. 相关文档

产品所需的其余相关文档，例如产品市场需求说明书（MRD）、产品功能介绍 PPT、产品规划书。

7. 附件

将产品需求的 Demo 作为附件，可以是原型图或者产品手稿等。

1.6　程序员的工作法则

如何衡量程序员的工作成效一直是个有争议的话题，而提供一些法则来提升程序员的工作成效已成为业界的共识，也是程序员的基本素养。

1.6.1　工具化法则

持续集成是一种软件开发实践，团队开发成员经常集成他们的工作，通常每个成员每天至少集成一次，这也意味着每天都可能发生多次集成，每次集成都通过自动化构建（包括编译、发布、自动化测试）来验证，从而尽快发现集成中的错误。许多团队发现这个过程可以大大减少集成的问题，能够让团队更快地开发内聚的软件。

先来看这样一个案例。某团队在具体部署 SystemP 的持续集成时发现一个问题：SystemP 重度依赖下游系统 SystemQ，而 SystemQ 系统开发环境的不稳定导致 CI 经常大量失败。该团队做了一个工具，该工具能够自动采集对下游系统成功调用的结果，并将其转换成缓存文件，后续的 CI 都使用该缓存文件，自动完成对下游的隔离，这极大提升了 CI 的稳定性。解决方案如图 1.8 所示。

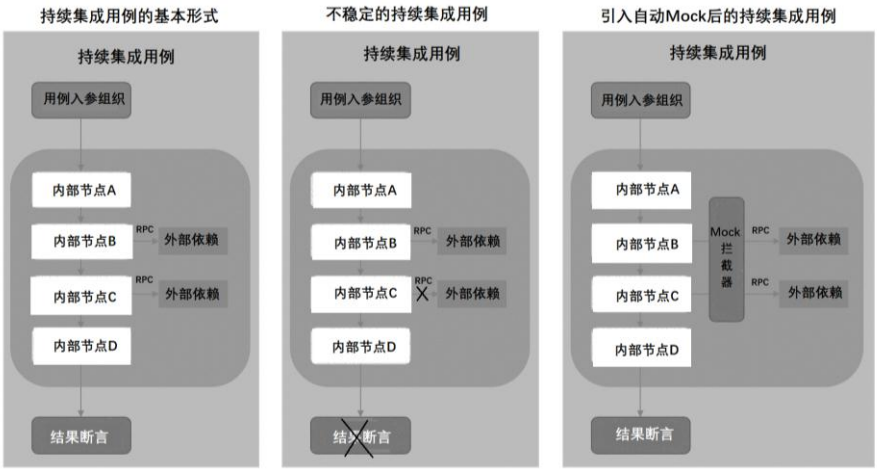


图 1.8

在图 1.8 中，当内部节点 C 依赖下游系统（比如 SystemQ）时，为了隔离下游系统的影响，该团队开发了一个 Mock 拦截器。Mock 拦截器的实现思路如图 1.9 所示，测试用例会对 Mock 数据是否存在进行判断，如果存在相关数据，则直接从缓存中获取。

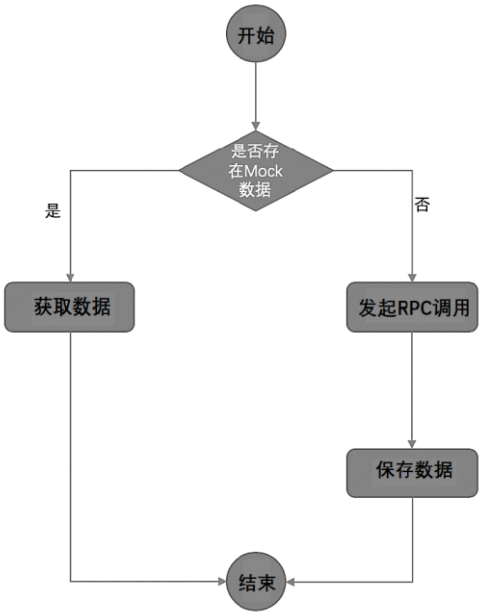


图 1.9

再来看一个案例。在 SOA 架构或者微服务架构中，联调是一件比较麻烦的事情，手工联调需要上下游多个系统约定时间，一起配合，好一点的做法是下游系统自己构造本地请求来模拟 RPC 调用。但是笔者遇到过这样一个系统，这个系统是一个规则决策类系统，上游系统大约有 20 个，各自的场景用例包括参数的不同组合达到 10 万这个级别，构造本地测试用例的覆盖难度极高，且不易于持续维护。于是开发人员做了一点创新，思路也比较简单：做一个能够自动打印历史请求的工具，然后通过工具平台将打印的请求重发到指定的服务器上，提升研发效率。

如图 1.10 所示，上游系统在联调时调用了服务接口，这时工具会进行请求拦截，把上游请求序列化并保存下来。在下游要触发验证时，就可以通过重发工具平台查找对应的日志，完成输入的触发。另外，在做回归测试时，采用类似的方法就可以从线上环境的用户真实访问日志中提取用例，并将其导入线下用例环境中。

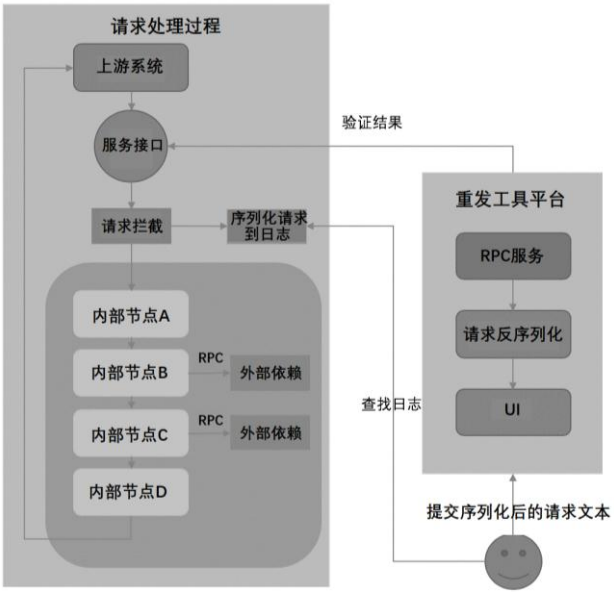


图 1.10

我们接着看一个系统迁移案例，系统迁移往往涉及将多个烟囱型架构迁移到平台型架构，或者将若干类似职责的系统统一到一个平台。一种常规做法是先观察和对比，再逐步切流，最后下线老系统。笔者所在的项目团队就曾开发过支持新老系统切流对比的工具。

为了确保新老服务结果的一致性和正确性，我们开发了一套切流对比工具，能够自动

将一个接口的流量复制一份给另一个对应的接口，在执行完成后，对两个接口的结果进行详细对比，验证其正确性。在业务场景复杂的业务系统上若没有类似的工具保障，则很难保证不出问题。该工具的实现如图 1.11 所示。

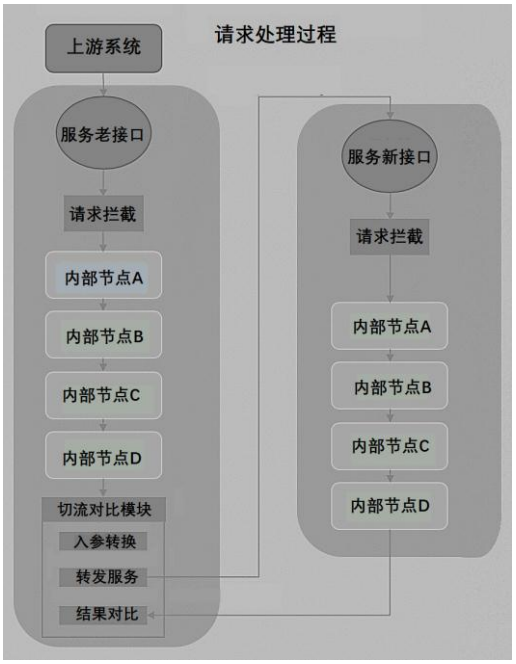


图 1.11

随着工具化思路的深入人心，各个团队都会构建、处理特定的问题，或者相对宽泛领域的问题。这时为了避免重复建设，应该进行相应的治理和统筹。一些似曾相识的场景如下。

◎ 场景一：研发人员小李需要解决分布式系统的用例沉淀的问题，他做了一个线上线下日志迁移的工具。

◎ 场景二：研发人员小周为了解决联调成本高及时间要求的问题，开发了前文介绍的重发工具平台。

◎ 场景三：在 SOA 架构下排查问题时，需要登录不同的系统去查看对应的日志文件，这样成本很高，某工具团队开发了一个基于 traceId 的日志关联工具。

◎ 场景四：测试工具团队开发了一个叫作“望眼镜”的工具，用于快速定位 SOA 架构下系统复杂依赖关系中 error 的传递情况，并用时序图展示。

◎ 场景五：在优惠场景下，由于业务的复杂度、优惠规则的复杂度等各种原因，用

户对于一个平台的支付工具使用存在认知偏差，对于支付的优惠券、红包和支付顺序可能存在疑惑，这时他们往往通过客服渠道反馈。客服人员一般求助产品经理或者技术人员，最终技术人员通过日志及数据库结合类似的 traceId 进行排查。由于类似的查询场景比较高频，所以技术同学研发了一个平台，将查询的结果翻译成客服人员可以明白的业务语义内容，供客服人员自助查询。

◎ 场景六：工具团队开发了一个故障排查模块，因为在 SOA 架构下定位问题基本靠口头沟通，比如交易大盘下跌了，各变更系统负责人都会去查看自身系统的错误情况，然后自己关联和思考故障的成因。那么，能不能把这些经验通过一系列规则脚本实现呢？

通过以上 6 种场景，我们可以总结出表 1.4，从该表中可以看到工具化解决问题的个性与共性。注意：以上场景其实只是冰山一角，对于上千人的团队，各种差异化的场景必然数不胜数，因此统一规划工具势在必行。

表 1.4

工具的名称	目标定位	研发团队	适用环境	备　注
线上用例迁移工具	从“脱敏”的日志中提取有效测试用例应用于回归测试	业务团队 A	从线上到线下	
重发工具平台	录制上游系统调用日志，重发调试	业务团队 B	线下	
日志关联工具	通过 traceId 查询多个跨系统的日志文件	工具团队	线上	
望远镜	快速定位 SOA 架构下系统的复杂依赖关系中 error 的传递情况，并用时序图展示	测试工具团队	线下	
业务优惠查询工具	如场景五的描述	业务团队 C	线上	
故障排查模块	如场景六的描述	工具团队	线上	

通过以上分析可以得到一些阶段性结论：故障排查工具用于线上的即时故障定位，对日志数据不用单独存储，重点在于快和准，这个工具是单独的一套架构体系；而“望眼镜”这个工具如果也是瞬间定位，那么完全可以复用故障排查模块的核心能力；系统调用时序图只是一个增强的特性。

结论：对于统一的工具平台，可以考虑不同工具的层次，使其能复用的尽量复用，并且注意平台的开放性，比如插件化能力；对于目标和实现方案差异很大的平台，则可以分别建设，比如业务优惠查询工具需要处理几天甚至 1 个月前的用户投诉，从日志中提取、转换和专门存储数据。故障排查模块则在于解决故障的快速定位问题，对于超过 1 个小时

的定位无法容忍，日志文件的存储时间默认为 7 天，不需要从日志中提取目标数据。这是这两个工具的最大差异，对其分别建设为宜。

1.6.2 自动化法则

自动化法则指将能自动化的都自动化。程序员为了提升效率，可以借助很多的工具来完成很多重复工作，下面就是借助工具完成自动化办公的一些案例。

1. 查询和定位自动化

搜资料、查单词、定位并打开文件、打开网址、打开书签、打开计算器及其他 App，这些通常是我们在办公过程中要做的事情。虽然这些事情都很简单，但是我们可以使用一个工具来更高效地做这些事情，这个工具就是 Alfred。Alfred 是 Mac 系统上的一个专注于效率提升的著名应用，通过这个工具简单配置很多流程化的操作，就可以用很简单的命令实现自动化操作。

◎ 在想搜索一个词汇时，直接通过快捷键调出 Alfred 输入框，输入“baidu 搜索词”即可。
◎ 若突然忘记一个单词的中文意思是什么，在 Alfred 中直接输入单词就可以知道其含义。
◎ 定位文件、打开文件是 Alfred 提供的基础功能，它能让你无须打开文件浏览器便可定位或打开文件，只需调出 Alfred 的输入框（默认快捷键是 Command + Space）输入对应的文字即可。在打开文件时直接输入要打开的文件或文件夹的名称，Alfred 便会将搜索结果显示出来，我们可以用 Command+数字进行结果选择。在定位文件或文件夹时则需要先输入“find 关键词+空格+对应的名字”。
◎ 在想打开对应的网址、书签或 App 的时候，无须打开浏览器，直接通过 Alfred 搜索即可。

2. 项目开发环境

如果你还在使用 Eclipse，那么你一定要试试 Intellij IDEA。围绕项目创建（import 项目）、代码开发、Debug、单元测试，这个 IDE 可以在很大程度上提升开发效率。下面是几个屡试不爽的功能。

◎ Maven 项目不需要执行 mvn eclipse:eclipse 等命令就可以直接 import。

◎ 在新增 pom 依赖配置后，自动下载依赖的 jar 文件。
◎ 对无源码的 jar 文件自动反编译，并且支持一键下载对应的源码（如果有的话）。
◎ 借助于一些插件可以自动生成代码，在保存后自动格式化代码、自动生成注释等。

3. Web 测试

有时测试一个 Web 页面的功能，需要重复输入、单击按钮和切换页面。借助于 Selenium，我们可以录制操作过程，自动生成测试脚本，然后实现相同功能的自动测试。

4. 部署、运维

我们在部署过程中有很多时候会借助 Shell 脚本自动完成一些任务。比如，在项目部署发布的过程中，往往需要先拉取代码（pull.sh），然后编译（build.sh），最后部署（deploy.sh）。在改动代码后还需要进行这几步。我们可以将这三个命令放到一起来执行，即./pull.sh&./build.sh&./deploy.sh，并将上面的命令写在一个 Shell 脚本里，这样就可以在服务器的任何地方执行 pbd 命令，来完成刚才的几步操作。

5. 测试

自动化作为有效的提效手段，是每个技术团队都在不断追求的目标。测试自动化的核心问题有两个：一个是分层，另一个是稳定。分层的思想体系在自动化测试建设比较成熟度的阶段，对一些小的日常发布完全可以做到无人值守，而对一些中大型的需求需要全面进行回归测试和质量兜底，要达到这个目的，通过一种自动化手段往往很难做到。比如单元测试或者接口测试做到了方法级别的粒度，但是在全链路层面无法涵盖；线上自动录制回放虽然通过引入线上流量进行测试，但是在新功能方面有比较大的局限。对于整个质量保障体系而言，多层次、不同粒度的测试自动化相互协同配合是个不错的选择，这也是建立单元测试、组件测试、接口测试、端到端全链路测试、录制引流回放、线上影子验证等不同维度测试的初衷。

稳定是持续运行的基础。对于一线研发人员而言，排查 CI 失败的脚本一定会有不愉快的回忆，甚至希望直接手动验证，而不要再排查繁杂的测试数据。这就是自动化的弊端，它将前人的测试经验代码化，但是也屏蔽了逻辑的细节，测试需要稳定、自动化地进行，而不是半自动化地进行。这符合引入测试成熟度的初衷：一方面评价代码覆盖，另一面评价 CI 的通过率。

工具化法则、自动化法则、加速法则都来自一线痛点。比如在 SOA 系统中，系统的

依赖数量较大，在每次进行开发环境测试时，都需要修改很多配置才能顺利进行。例如，配置 RPC 路由地址、配置消息中心绑定消息的接收者和发送者等。

解决办法为一键配置开发沙箱环境，对每个应用都配置一个统一的项目 ID，然后自动将有相同项目 ID 的应用拉取的代码及获取的虚拟机器资源，配置成一个虚拟的沙箱体系。在同一个沙箱体系内，消息不会被发送到外部，调用也在沙箱内完成，完全隔离。

当然，解决方案不止一种，但极度的工具化，尤其是把高频的重复劳动自动化，可以大大提升效能，并让程序员获得满足感和成就感。

1.6.3 关于文档的问题

要不要写文档，文档写多少？有位经验丰富的朋友说：“应该将文档整理成思维树，标注定义、关键描述、关键词、关联词和详细的文档链接。这对新人来讲更容易找到切入点，并快速了解项目或系统的全貌，以及可能与自己未来工作相关的上下文（内涵、外延）是什么。传统的文档管理方式往往使新人在浩如烟海的文档中迷失。

关于文档的问题，笔者有如下几个观点。

◎ 我们需要保留项目研发过程中的关键文档，比如需求和系统分析文档，这些是项目中不可或缺的文档。
◎ 一个系统或平台都需要有 Core 文档，比如领域模型、主体架构等，由于这部分文档的更新并不频繁，所以可以定期维护。
◎ 用例即文档，使验收测试及接口测试等保持稳定，是研究细节和用户场景的入手点。
◎ 提倡活文档，具体的推荐做法是接口文档通过接口声明生成，接口声明对于每个参数都会有说明。在 Swagger Editor 中，我们可以基于 YAML 语法定义 RESTful API，它会自动生成一篇排版优美的 API 文档，在 API 改变之后，API 文档也会随之改变。

1.6.4 关于 YAGNI

YAGNI（You Ain’t Gonna Need it，你不会需要它）的意思是：如无必要，别增复杂度。Cunningham & Cunningham（http://wiki.c2.com）将其解释为：即使你非常确信自己将来需要某个特性，也不要现在就去实现它。在很多情况下，你会发现或许最终自己不需要它了，或者真正需要的特性与之前预期的有很大出入。遵循 YAGNI 实践有如下两个主要原因。

◎ 你节约了时间，因为你避免编写了最终证明不必要的代码。
◎ 你的代码质量更高了，因为你让代码不必为你的“推测”所污染，而这些“推测”最终可能或多或少有些错误，但此时这些错误已牢牢地依附在你的代码中了。

Martin Fowler 进一步表示：当我们在考虑推定特性时，很有可能我们是错的。在这种情况下，推定特性的一个很明显的成本就是整个构建过程的成本，也就是对这个在当下没有用处的特性进行分析、编码及测试所耗费的精力。他同时表示，假设我们对这个需求的理解恰好是正确的，则即使在这种比较理想的情况下，创建这个推定特性同样会带来两种巨大的成本，第 1 种是软件价值的延误成本，第 2 种是延续这一特性所带来的成本。

这里有个思辨：如果对一个问题有两种解决方案，第 2 种解决方案明明有更好的“增强扩展性”，而两种解决方案的实现成本相差无几，难道我要墨守 YAGNI 原则吗？

下面以两个案例来说明这个问题。

◎ 第 1 个案例，某团队去年接到一个钱包积分需求，经过分析，他们的设计不仅仅实现了钱包积分，还实现了商户通用积分的生命周期管理。因为后者有扩展性，并从业务上依稀感受到未来或许有这样的必要，并且经过评估，实现成本没有增加太多，该团队就果断采取了商户通用积分的实现方案。过了 1 年，果然，类似的需求来了……
◎ 第 2 个案例，某团队在 2014 年年底规划了一个券平台，应对此前的各种红包、优惠类业务的烟囱架构问题。这个平台的提出基于若干需求背后的核心领域抽象，这是他们在几年前做红包系统时万万不能想到的。当时他们看到的需求就是红包，对应的领域对象也是红包实体、红包模板及规则等，也就是说对问题域的本质认知随着业务的发展而发展。延迟设计决策是非常重要的，对这个度的把握很难，若没有把握好，用不了两年就得返工。

这里得出结论：YAGNI 并非不做预设计，而是在成本、复杂度之间做权衡。我们现在需要建立一些机制（比如抽象、约束）应对它。注意，只是建立机制，在实现层面仍然按当前刚刚好的功能实现，除非能找到额外复杂度非常小的方案把预期的变化也覆盖了。

1.7　程序员的工具箱

Perl 之父 Larry Wall 曾在 *Programming Perl* 一书中提到：程序员的三个美德是懒惰、不耐烦和傲慢。懒惰，是程序员美德的第一要素。

Larry Wall 所说的“懒惰”，并不是安于现状和不思进取，而是付出最少的时间或者精力来达到同样甚至更好的目标。“懒惰”的程序员会尽量使自己的代码既实用又有很好的可读性，这样可以节省后面的很多维护成本；还会尽力完善代码中的注释及文档，以免别人问自己太多问题，更擅长使用各种工具，从方方面面提升自己的效率。

本节将要介绍的多种实用工具，就可以帮我们全方位武装成“懒惰”的程序员，例如可帮我们提升工作效率的硬件、操作系统和软件。

1.7.1 可帮我们提升工作效率的硬件

这里盘点一下那些可以提升程序员工作效率的硬件设备。

1. 显示器

笔记本电脑基本上是程序员开发的标配，那么要不要给自己再多配置一台显示器呢？配置一台多大的显示器呢？

在 StackOverflow2018 开发者调查报告中显示，有 51.1%的开发者使用两个显示器，有 17%的开发者使用三个以上显示器。

美国犹他州立大学的研究人员也测试了工作效率与不同电脑配置的关联性。他们让受访者完成同样的工作，测试结果如下（http://dwz.cn/Kf579）：

◎ 使用 24 寸显示器工作的人比使用 18 寸显示器工作的人在工作效率上要高 52%；
◎ 使用两个 20 寸显示器工作的人比使用单个 18 寸显示器工作的人在工作效率上要高 44%；
◎ 当显示器达到 26 寸时，受访者的工作效率开始下降。

所以，我们基本可以得出结论：额外配置一台 24 寸左右的显示器可以提升工作效率。因为作为程序员，我们可能要同时和 IDE、Linux 命令行及浏览器等打交道，有多台显示器可以帮我们减少切换时间。

2. 外设

键盘和鼠标等外设对于程序员来说，其重要性不言而喻。在 StackOverflow2018 开发者调查报告中显示，有超过 52%的开发者认为人体工程学键盘和鼠标可以使自己在繁忙的工作中感到一点点舒适感。

关于键盘和鼠标，笔者认为，一个好的机械键盘或电容键盘加一个比较灵敏的无线鼠标，不仅可以令一个程序员的工作效率得到很大提升，还可以在一定程度上起到愉悦心情的作用，尤其是通过调试解决一个 Bug 后，最后敲击键盘的那一刻。

1.7.2　可帮我们提升工作效率的操作系统

目前，主流的操作系统共有三种：Linux、Windows 及 Mac OS。关于操作系统的使用及选择，StackOverflow 在 2018 年也做过类似调查（Developer Survey Results 2018，https://insights.stackoverflow.com/survey/2018），结果显示：主要使用 Windows 操作系统的开发者占了 49.9%，主要使用 Mac OS 操作系统的开发者占了 26.7%，主要使用 Linux 操作系统的开发者占了 23.2%。从数字上来看，虽然使用 Windows 操作系统的开发者占了一半左右，但另一项关于程序员最想使用的系统平台的调查结果显示，有 76.5%的开发者期望使用 Linux 操作系统，有 63.9%的开发者喜欢并且期望使用 Mac OS 操作系统，有 61.2%的开发者期望使用 Windows 操作系统。

对于以上三种操作系统，笔者在自己的开发生涯中均使用过，目前使用 Mac OS 作为主要的操作系统进行开发。由于开发者开发出来的应用基本都部署在 Linux 服务器上，而且 Linux 操作系统提供了很多 Shell 命令，可以快速地进行构建、编译、执行、排查问题等，所以在日常的纯代码开发、交付及运维方面可以大大提升效率。但是，程序员又无法避免地要和各种软件打交道，比如开发工具、浏览器、画图工具等，这时提供图形界面的 Windows 操作系统就可以很好地支持这些图形化软件了。Mac OS 作为一款基于 UNIX 的图形化操作系统，就兼容了 Linux 和 Windows 操作系统的优点，既可以运行 Shell 命令，又可以进行图形化操作，当然，Mac OS 操作系统也有一些缺点，比如支持的软件不够多、价格昂贵等。

笔者觉得，程序员选择操作系统的优先级顺序应该是 Mac OS、Linux、Windows。

1.7.3　可帮我们提升工作效率的软件

这里再介绍一些可以帮助开发者提升工作效率的软件，主要包括开发工具、版本控制工具、自动化构建工具和画图工具等。

1. 开发工具

对于开发者，尤其是 Java 开发者来说，使用开发工具编写代码真的可以大大提升工作

效率，因为现在很多 IDE 都有很强大的工具，不仅提供了代码补全、错误提示、自动编译等功能，还提供了各种插件，方便和其他工具融合，能大大提升写代码的效率及代码质量。

目前市面上主要有两款 Java 代码开发工具：Eclipse 和 IntelliJ IDEA，这两款开发工具之争由来已久，不过最近几年，IntelliJ IDEA 逐渐撼动了 Eclipse 的霸主地位，成为开发者的首选开发工具。在 2013 年的 Google I/O 大会上，Google 推出了新的 Android 集成 IDE “Android Studio”，其最大的转变就是从 Eclipse 切换到了 IntelliJ IDEA。

笔者也经历过从 Eclipes 转向 IntelliJ IDEA 的过程，相比之下，Intellij IDEA 在某些方面确实比 Eclipse 更加出色一些，配合 Mac OS 操作系统上的快捷键，真的可以在开发效率上提升一大截。

关于开发者使用的 IDE，Jet Brains 做过一项调查，结果显示：有 59%的开发者使用 IntelliJ，有 30%的开发者使用 Eclipse，只有不到 5%的开发者使用 Sublime 和 Vim 等普通文本编辑器。

2. 版本控制工具

程序员在协作开发时不可避免地要使用版本控制工具对代码进行管理，版本控制工具主要用于存储和追踪被管理的文件的修改历史，是软件开发者的必备工具。

目前市面上比较成熟的版本控制工具有 CVS、Visual SourceSafe、PVCS、Subversion（SVN）和 Git 等，SVN 和 Git 的使用较为广泛。

无论是 SVN 还是 Git，都提供了很好的版本控制功能，例如对代码的统一管理、冲突解决、代码权限控制、分支开发和合并等。笔者认为，开发者应该熟练掌握 SVN 和 Git，比如代码拉取、代码提交、代码合并和冲突解决等，尤其要掌握一些 Shell 命令，这可以节省大量的时间。

虽然目前有很多开源代码及很多公司的内部代码都倾向于使用 Git 进行代码托管，但我们不能保证所遇到的所有代码都使用了 Git。公司规模越大，使用 SVN 进行代码托管的可能性便越大，毕竟迁移也是需要一定成本的。

3. 自动化构建工具

自动化构建指自动创建软件组建的一组进程，包括将计算机源代码编译成二进制码、将二进制码包装成软件包并对其进行自动化测试。很早以前的自动化构建都是通过创建 Make 来完成的，后来发展为 GNU Make。随着时间的推移，如今的软件开发主要使用更

优秀的自动化构建工具来完成，例如 Java 世界中的三大构建工具 Ant、Maven 和 Gradle，如今，Ant 几乎要退出历史的舞台了。

Maven 和 Gradle 之所以能够赢得众多程序员的青睐，主要是因为它们在依赖管理、冲突解决、项目构建、项目结构管理和插件机制等方面的出色表现。

4. 画图工具

所谓一图胜千言，无论是自己梳理思路，还是给别人讲解，图案都是很好的形式。而且，程序员还经常要做一些 PPT 等，有了这些画图工具，就能大显身手。

1）思维导图——XMind

人类大脑的最自然思考方式是放射性思考，也就是我们常说的发散性思维。我们通常在思考一个问题的时候，都是以一个思维点为起点，然后不断进行发散式展开的。如果我们将这个思考过程通过图形化的方式表达出来，就是一张思维导图。思维导图可以有效地把思维模式表现出来，有利于人脑的扩散思维的展开。如图 1.12 所示是一张关于“提升工作效率的软件”的思维导图。

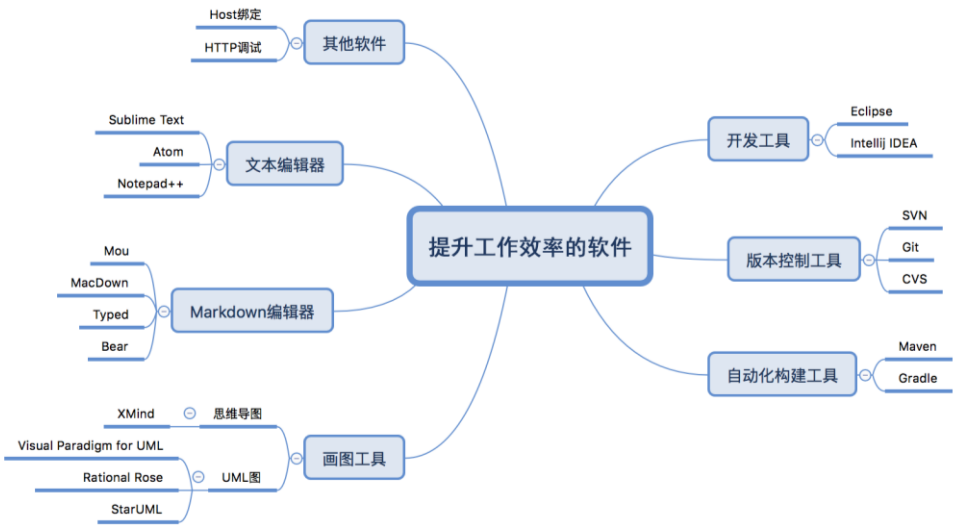


图 1.12

XMind 是一款非常实用的商业思维导图软件，采用 Java 语言开发而成，具备跨平台运行的性质，且基于 EclipseRCP 体系结构，支持插件通过编写 XML 清单文件扩展系统定

义好的扩展点，帮助用户在真正意义上提高生产效率，被著名的互联网媒体 Lifehacker 评选为“最佳头脑风暴和思维导图工具”及“最受欢迎的思维导图软件”。

XMind 的程序主体由一组插件构成，包括一个核心主程序插件、一组 Eclipse 运行时插件、一个帮助文档插件和一组多语种资源文件插件，对 Eclipse 用户非常友好。

XMind 不仅可以绘制思维导图，还可以绘制鱼骨图、二维图、树形图、逻辑图和组织结构图（Org、Tree、Logic Chart、Fishbone），并且可以方便地在这些展示形式之间进行转换。用户可以导入 MindManager、FreeMind 数据文件，灵活定制节点外观、插入图标、丰富的样式和主题。其输出格式有 HTML 和图片。

2）UML 画图工具

UML（Unified Model Language）即统一建模语言，又被称为标准建模语言，是用来对软件密集系统进行可视化建模的一种语言。开发人员在做详细设计时，免不了要和各种 UML 图打交道，例如用例图、类图、对象图、时序图、活动图和状态图等。

简单来说，开发人员在写代码之前就需要了解需求，在这之后要对软件系统进行建模，将抽象的语言描述转换成可视化的模型。而在建模过程中需要一种标准的语言，这种语言就是 UML。

目前市面上有很多 UML 画图工具可供选择，它们均能满足基本的画图功能，下面简单列举几种。

◎ Rational Rose：是 Rational 公司出品的一种面向对象的统一建模语言的可视化建模工具，主要用于可视化建模。

◎ Visual Paradigm for UML（VP-UML）：是一种功能强大、跨平台、使用便捷、直观的 UML 建模和 CASE 工具，它可以被整合在其他 CASE 工具或者其他 IDE 工具中。Visual Paradigm for UML 支持 UML 建模、数据库建模、对象关系映射、逆向工程和 Java 双向工程等，功能异常强大。

◎ StarUML（SU）：是一款开源的 UML 开发工具，支持绘制用例图、类图、序列图、状态图等 9 种常用的 UML 图，具有发展快、灵活、可扩展性强等优点，唯一的不足之处是在反向工程时只能生成类图，不能生成类之间的关系。

5. Markdown 编辑器

Markdown 是一种轻量级标记语言，创始人为约翰·格鲁伯（John Gruber），允许人们

使用易读易写的纯文本格式编写文档，然后将文本格式转换成有效的 XHTML（或者 HTML）文档。

作为程序员，我们除了要具备编程的硬技能，还要具备写作的软件，而首选的写作方式就是 Markdown。目前，很多大型博客网站都开始支持使用 Markdown 写文章，这无疑是对程序员的一种示好行为，著名的代码托管网站 GitHub 也支持使用 Markdown 语法来编辑 README 文件等，如图 1.13 所示。



图 1.13

关于 Markdown 的编辑器，目前市面上也有很多选择，主要有 Mou、MacDown、Typed 和 Bear 等，还有很多在线的 Markdown 编辑器，在后面的在线工具章节中会详细介绍。

还需要提及的一点是，其实使用哪种 Markdown 编辑器并不很重要，重要的是程序员应该掌握常用的 Markdown 语法。这样写出来的文章或者文档，可以在任意支持 MD 语法的平台上查看。

6. 文本编辑器

前面介绍过 Java 开发常用的 IDE，但是在某些情况下我们仅需要打开一个单独的文本文件，比如打开一个 XML 格式的文件、一个 JSON 格式的文件等，这时一款好用的文本编辑器就派上用场了。

Sublime Text 是一款流行的代码编辑器软件，可运行在 Linux、Windows 和 Mac OS X 等操作系统之上，是许多程序员喜欢使用的一款文本编辑器软件。它具有代码高亮、语法

提示、自动完成且反映快速的编辑器软件，不仅界面华丽，还支持插件扩展机制，如图 1.14 所示。

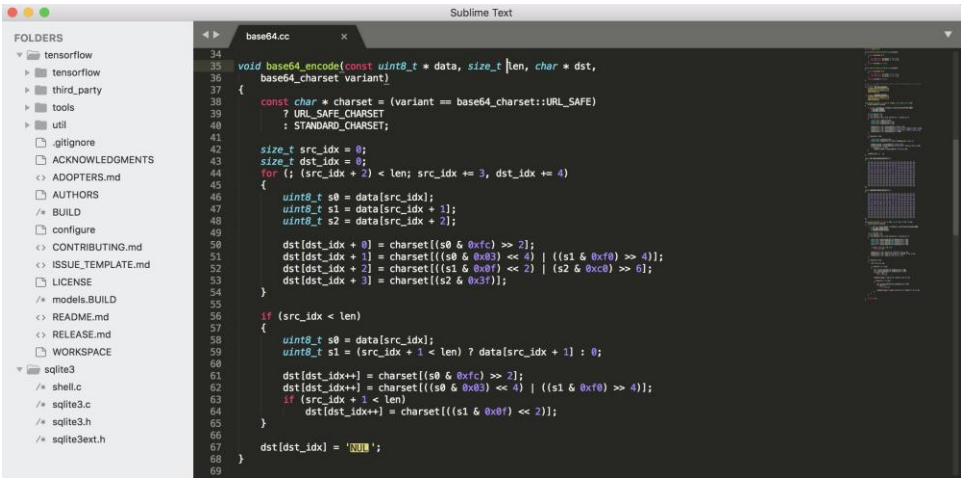


图 1.14

Atom 是 GitHub 专门为程序员推出的跨平台的文本编辑器，因为有简洁、直观的图形界面而受到了广大程序员的青睐，而且它有很多有趣的特点：支持 CSS、HTML、JavaScript 等网页编程语言；支持宏，自动完成分屏功能，集成了文件管理器；支持 Mac OS、Windows 和 Linux 操作系统，支持 Node.js 所写的插件，并内置由 GitHub 提供的 Git 版本控制系统。多数的延伸包皆为开放源代码授权，并由社区建置与维护。

Notepad++也是一套被广大程序员选用的纯文本编辑器，基于同样开放源码的 Scintilla 文本编辑组件研发而成，整个项目起初被托管于 SourceForge.net，曾经两度获得 SourceForge 社区选择奖——最佳开发工具。Notepad++在 2010 年 6 月被托管于 TuxFamily，现被托管于 GitHub。

7. 其他软件

除了以上介绍的一些必备软件，还有些常用工具也是程序员在日常工作中必不可少的，例如 Host 绑定、HTTP 调试等。

1）Host 绑定

在日常开发工作中，我们对 Host 绑定肯定不陌生，因为我们几乎每天都要和它打交

道。我们可能经常要在不同的 Host 绑定之间来回切换，尤其是在 Java Web 开发中，在本地开发测试时可能使用的是一套 Host 绑定，在测试环境部署之后提交给测试人员进行测试时又要使用另一套 Host 绑定。在测试之后，要经历预发布、灰度发布等，都需要一套新的 Host 绑定。采用好用的 Host 绑定工具可以大大节省 Host 绑定切换的时间。

SwitchHosts 是一个非常方便、快捷的 Host 绑定切换工具，免费、开源，支持预发高亮、方案多选、单击行号快速切换注释、系统菜单栏快捷切换、方案导入导出等功能，如图 1.15 所示。

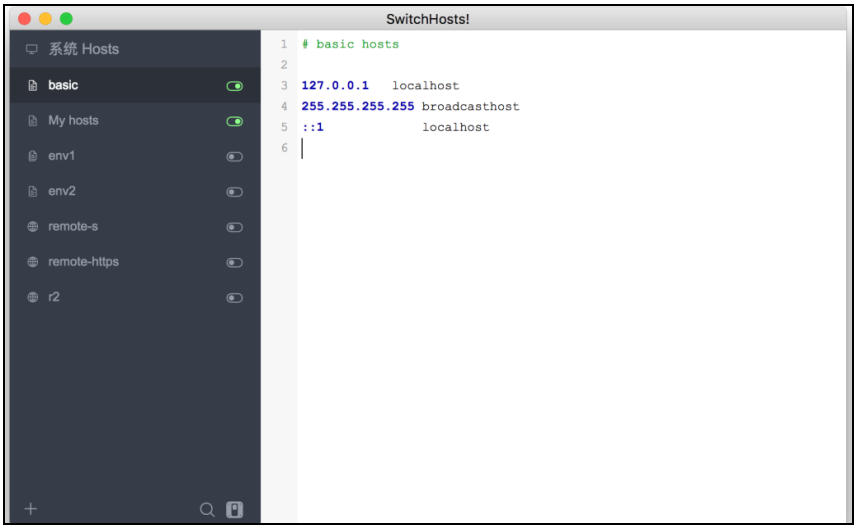


图 1.15

2）HTTP 调试

在日常的 Web 开发中，我们除了会和 Host 绑定打交道，还经常会和 RESTful API 打交道，有时我们需要一个工具来帮我们发送 HTTP 请求，或者说是模拟 HTTP 请求，比如帮我们发送 POST 请求等，这时就需要一款可以模拟发送 HTTP 请求的工具。

Postman 是 Google 开发的一款功能强大的网页调试与发送网页 HTTP 请求，并能运行测试用例的 Chrome 插件。除插件外，也提供软件下载。它可以模拟各种 HTTP 请求，从常用的 GET、POST 到 RESTful 的 PUT、DELETE 等，甚至可以发送文件及额外的 Header 等。Postman 可以将 Response 内容的格式自动美化，将 JSON、XML 或者 HTML 都整理成我们可以阅读的格式。Postman 还支持编写测试脚本，可以快速检查 Request 结果，并

返回测试结果。总之，Postman 是一款非常不错的可以用来做 HTTP 调试的软件。

1.7.4 在线工具

前面介绍了很多常用软件，接下来介绍一些在线工具，我们无须安装这些在线工具，可以在浏览器上直接使用，非常便捷。

1. 搜索类在线工具

本节讲解几种搜索类在线工具。

Grepcode（http://grepcode.com/）是一个面向于 Java 开发人员的网站，在这里你可以通过 Java 的 projects、classes 等关键字在线查看对应的源码。通过 Grepcode，还可以在线查看很多不同版本的源码，并下载相关的 jar 包、源码 jar 包和 Doc 文件。同样，我们可以使用 xxx-1.1.1.jar 类似的名字直接找到对应的 jar 包并下载。

SearchCode（https://searchcode.com/）是一个源码搜索引擎，目前支持从 GitHub、Bitbucket、Google Code、CodePlex、SourceForge 和 Fedora Project 平台搜索公开的源码，如图 1.16 所示。

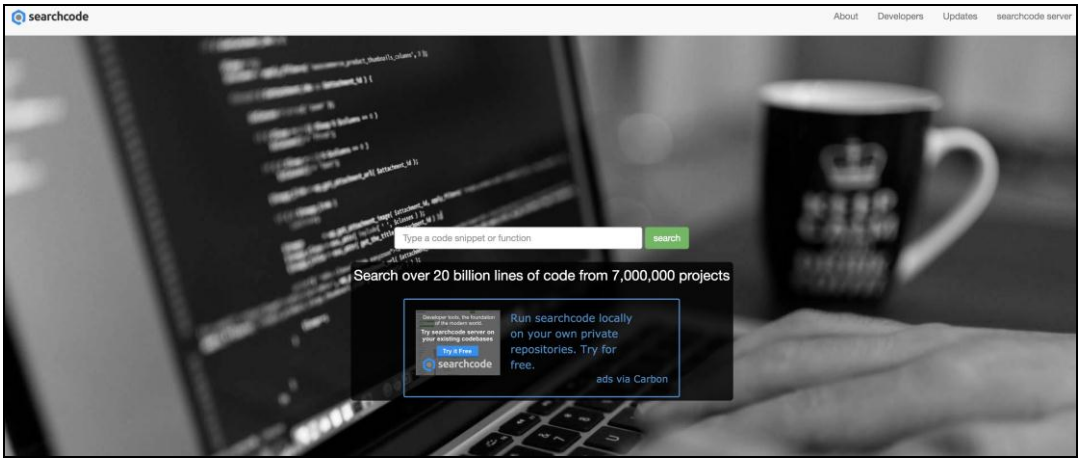


图 1.16

对于 mvnrepository（http://mvnrepository.com）就不详细解释了，它用于查询 Maven 的 gav（groupId、artifactId、version）等信息，如图 1.17 所示。

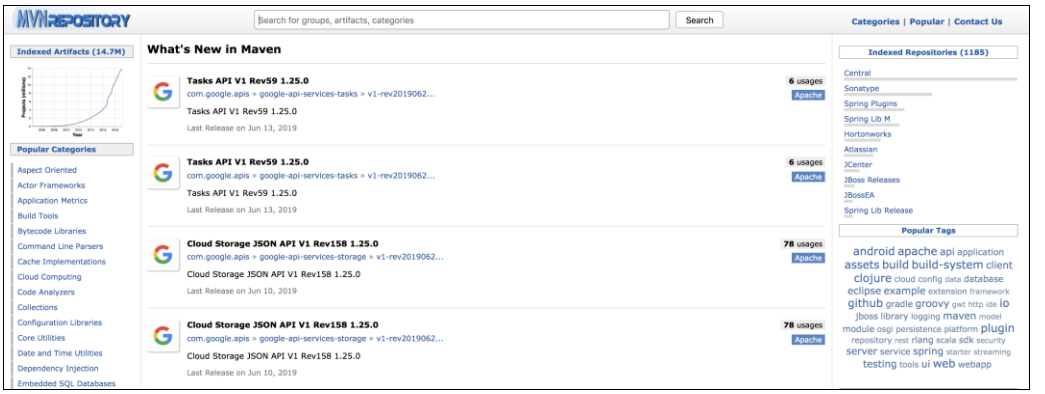


图 1.17

2. 生成类在线工具

下面讲解各种生成类在线工具。

1）用 SQL 自动生成 Java 代码

AutoJCode（http://www.autojcode.com/code/sql2class.jsp）可以从 SQL 的建表语句中生成一个 DO 类。很多时候，我们在有了数据库的建表语句之后，会希望快速生成与之相对应的 POJO，这时这种自动生成工具就派上用场了。

2）用 JSON 生成 Java 类

JSON 是目前 Java Web 中数据传输的主要格式，我们在很多时候都会有将 JSON 转成 Java 对象的需求。有时合作方会提供一个 JSON 样例，需要我们自定义 Java 类，这时这个工具就派上用场了。BeJSON（http://www.bejson.com/json2javapojo）就是一款比较好用的工具。

3）Corn 表达式的生成

Cron 一般用于配置定时任务的执行，但是要想一次性地把一个 Corn 表达式配置好，确实很难，需要我们记住其语法。有一些在线工具可以提供图形化的界面，只要输入想要定时执行的周期等，就可以自动生成 Corn 表达式。推荐使用 pdtools 提供的 becorn（http://www.pdtools.net/tools/becorn.jsp）。

4）正则代码的生成

正则表达式，又叫作规则表达式，是计算机科学中的一个概念，通常被用于检索、替

换那些符合某个模式（规则）的文本。在使用正则表达式进行字符串过滤时，需要用事先定义好的一些特定字符及这些特定字符的组合，组成一个规则字符串，这个规则字符串用于表达对字符串的一种过滤逻辑。这个规则字符串的定义通常是比较麻烦和复杂的，也需要经过大量的测试和验证才能被采用。某些网站（如 http://tool.chinaz.com/tools/regexgenerate）提供了正则表达式的生成工具，可以一键生成身份证号、邮箱、手机号等常用的正则表达式。

3. 转换类在线工具

下面讲解一些转换类在线工具。

1）编码转换

编码问题一直困扰着开发人员，尤其在 Java 中更加明显，因为 Java 是跨平台的语言，在不同的平台之间及编码之间的切换较多。在计算机中提供了多种编码方式，常见的有 ASCII、ISO-8859-1、GB2312、GBK、UTF-8、UTF-16 等。有些时候，开发人员需要通过编码转换的方式来查看不同编码下的文件内容。

“站长工具”的编码转换（http://tool.chinaz.com/tools/unicode.aspx）功能比较全面，提供了 Unicode 编码、UFT8 编码、URL 编码及解码等功能。

2）时间戳转换

时间戳（英语：Timestamp）是指在一连串的资料中加入辨识文字，如时间或日期，用于保障本地端（local）资料的更新顺序与远端（remote）一致。

在 Java 中有很多地方都会用到时间戳，也经常会使用这种转换工具（http://tool.chinaz.com/Tools/unixtime.aspx）。

3）世界时间转换

世界时间转换，比较常见的是将北京时间转换成纽约时间、将纽约时间转换成洛杉矶时间等。Timebie（http://www.timebie.com/cn/easternbeijing.php）便提供了世界时间相互转换的功能。

4）加密、解密

加密、解密也是在 Java Web 中经常遇到的，有时我们需要验证加密算法是否正确，或者需要解密等，就需要一款在线工具（参见 http://tool.chinaz.com/tools/textencrypt.aspx）

来快速验证了。

5）单位换算

Convertworld（https://www.convertworld.com/zh-hans/）是一个比较全的单位换算的网站。笔者经常用它进行时间单位和货币单位的换算。

4. 检查类在线工具

下面讲解一些检查类在线工具。

1）JSON 在线验证及格式化

JSON 是一种由道格拉斯·克罗克福特构想和设计的轻量级的数据交换语言，该语言以容易让人阅读的文字为基础，传输由属性值或者序列性的值组成的数据对象。目前有很多编程语言都支持 JSON 格式数据的生成和解析。

在很多 Web 应用中都会使用 JSON 进行数据传输，所以，我们在开发过程中会经常接触 JSON 格式的文本，有时我们不确定这个文本是否完全符合 JSON 格式，或者想更清晰地查看这个 JSON 文本的格式关系，这就需要一款工具进行 JSON 格式的验证和格式化。

笔者用过很多 JSON 在线格式化工具，经过实践发现，Json.cn（https://www.json.cn/）比较不错，不仅支持 JSON 格式的验证及格式化，还可以将 JSON 格式压缩成普通文本等，如图 1.18 所示。



图 1.18

2）Diff Check

使用过 SVN 或者 Git 的人对 Diff Check 肯定不陌生，但有时候我们修改的文本内容并没有被版本控制工具管理起来，那么可以使用在线的网站查看文件的修改情况，比如 Diffchecker（https://www.diffchecker.com/），如图 1.19 所示。



图 1.19

3）正则验证

Java 开发人员对正则表达式肯定不陌生。“站长工具”提供的正则验证（http://tool.chinaz.com/regex）就比较不错。

5. 对照类在线工具

对照类在线工具如下。

◎ ASCII 对照表：http://tool.oschina.net/commons?type=4。
◎ HTTP 状态码：http://tool.oschina.net/commons?type=5。
◎ HTTP Content-type：http://tool.oschina.net/commons。
◎ TCP/UDP 常见端口参考：http://tool.oschina.net/commons?type=7。
◎ HTML 转义字符：http://tool.oschina.net/commons?type=2。
◎ RGB 颜色参考：http://tool.oschina.net/commons?type=3。
◎ 网页字体参考：http://tool.oschina.net/commons?type=8。

6. 其他类型的在线工具

下面讲解其他类型的一些在线工具。

1）在线 UML 制图

ProcessOn（https://www.processon.com/）是一个在线协作绘图平台，为用户提供了非常强大、易用的作图工具，支持在线创作流程图、BPMN、UML 图、UI 界面原型设计、

iOS 界面原型设计等。

2）在线代码运行

coderunner（https://tool.lu/coderunner/）可以在线运行 PHP、C、C++、Go、Python、Java、Groovy 等代码，在我们手头没有 IDE 又想执行一个小程序时是个不错的选择。

第 2 章

加速成长与学会学习

为什么要谈持续学习这个话题？因为知识总是会过时的，比知识更重要的是思维方式；另外，大部分人的聪明程度相差不大，有足够的练习和实践是必需的，终身学习才是成功者的必备法宝。

2.1 程序员如何加速成长

每一个程序员都希望自己加速成长，并适应这个飞速发展的行业。本节就来讲讲程序员如何加速成长。

2.1.1 积极主动

美国著名的管理学大师史蒂芬·柯维在自己的《高效能人士的七个习惯》一书中曾经说过：从依赖期走向独立期，第一个必须建立的习惯，也是最重要的习惯，就是积极主动。

每一个程序员的职业发展路径也必然要从依赖期走向独立期，而积极主动正是衡量一个独立开发者的重要指标。

对于一个程序员来说，主动可以表现在方方面面，比如主动申请承担一份有挑战的工作，主动选择使用新的开发框架，主动重构代码中不好的地方，主动向团队内的同事分享最近学习的知识点，主动优化代码中的性能问题等。

在很多大公司中，程序员们的分工越来越细致，从一开始的全栈开发，慢慢区分出运维工程师、开发工程师和测试工程师。慢慢地，开发工程师又区分出前端开发、后端开发和移动端开发。后端开发同样可以区分出业务开发、中间件开发；同样是业务开发，又可

以深入地划分成不同业务领域的开发，比如电商业务开发、支付业务开发、游戏开发，等等。随着分工的细致化，程序员更要积极主动地去了解其他人所做的事情，只守着自己的“一亩三分地”，被动地工作，是无法成为一个优秀的程序员的。

2.1.2　空杯心态

空杯心态，是一个心理学概念，来源于一个小故事：

一位佛学造诣很深的人，去拜访一位德高望重的老禅师，在被老禅师的徒弟接待时，他态度傲慢，自认佛学造诣很深，这个徒弟不配接待自己。后来老禅师十分恭敬地接待了他，并为他沏茶，在倒茶时，明明杯子已经满了，可是老禅师还在不停地倒茶。他不解地问：“大师，为什么杯子已经满了，还要往里倒茶？”大师说：“是啊，既然已满了，为什么还要倒茶呢？”意思是，既然你已经很有学问了，为什么还要到我这里求教？

空杯心态，就是时时刻刻把自己想象成“一个空着的杯子”。空杯心态并不意味着要否定自己过去，而是怀着一种放空的态度融入新的环境，对待新的工作和事物。

程序员在职业生涯中有很多场景都需要空杯心态。比如进入一个新的项目组，对新的项目组使用的技术并不了解，这时就要清空自己对原有技术栈的执念等，努力学习新的技术和知识。又比如进入一家新的公司，更要清空自己过往的工作成绩，以一个新人的心态来主动、虚心地学习。

2.1.3　选择合适的平台

有句话叫“选择比努力重要”，笔者有一个亲身体会。笔者在大学毕业后在电信行业做软件研发，在此工作 5 年之后发现成长缓慢。在转到互联网公司工作后，发现自己的五六年工作经验在互联网领域内未必能胜过一开始就在这个领域工作并积累了三年工作经验的人。

所以，笔者的一点浅见就是选择合适的平台，等我们成长慢时，就应该到下一个平台：换一个业务、岗位或者公司。主动换平台是不太容易的，因为我们可能已经习惯做某些事务，这种习惯就在舒适区内。

2.1.4 别怕犯错

很多程序员，尤其是新人程序员都很怕犯错误，导致自己畏首畏尾、不敢尝试。其实大可不必，无论多么成功的人都不可能不犯错误。

笔者刚刚入职阿里巴巴时，在一次百年阿里的培训课上，听到很多淘宝初期的程序员讲他们当年的故事，其中很重要的一部分就是每个人都会讲自己曾经犯过哪些错，导致哪些线上故障。这些人现在已经是各个事业部的架构师、技术负责人，他们也是从错误中摸爬滚打过来的。他们给新人很重要的一条建议就是：别怕犯错，最重要的是从错误中学习到什么。

阿里巴巴研究员毕玄就是一个特别爱总结和从错误中学习的人。他有一篇文章中专门深挖自己过去在系统设计上犯过的 14 个错误（https://yq.aliyun.com/articles/33077），并且他有个博客（http://hellojava.info/），主要用于记录自己在工作中遇到的线上问题及解决方案。这种不害怕犯错，并且善于从错误中总结和学习的态度在他的职业生涯中给了他很大的帮助。

需要注意的是，别怕犯错，不等于不敬畏错误。别怕犯错，鼓励的是别害怕尝试，别害怕犯高级错误，有很多低级错误还是要尽量避免的。

2.1.5 注意细节

前面提到过，无论多么伟大的程序员，都是不可能不犯错误的。我们能做的就是在问题发生之前尽量避免问题，在问题发生之后以最短的时间解决问题，并且在事后进行总结和沉淀，避免再次出现同样的错误。错误往往是隐藏在细节中的，在软件开发领域也符合经典的二八原则：80%的错误可能隐藏在 20%的代码中，这 20%的代码可能是业务逻辑中非常边缘化的细节问题。

其实细节就是我们的习以为常之处。比如在开发中，多考虑是不是会发生并发问题，在远程服务调用时多考虑发生了超时怎么办，在做数据的写操作时多考虑需不需要加锁或者做事务等。

2.1.6 时间管理

对于大部分程序员来说，工作时长一般都比较长，大部分会超过 12 小时，有的甚至

是 14～16 小时。但是，如果真的可以很好地管理时间，那么可能节省很多时间。很多人整天都很忙，手头的事情总是忙不完，其实是因为没有搞清楚该如何管理自己的额外时间，或者说不知道如何分配自己的时间。本节就来讲讲如何进行时间管理。

1. 主动管理时间

前面提到工作要积极主动，在时间管理上也一样，需要主动管理自己的时间。一个程序员一天可能有很多事情要做：写代码，参加各种会议及讨论，进行问题排查、项目管理、工作沟通、答疑等。如果没有很好地管理时间，那么留给写代码的时间可能就很少了，这也是很多程序员在晚上 6 点下班后才开始写代码的原因。

笔者在这方面有一些经验，可以供大家参考。

（1）关于会议：参加必要的会议，尽量避免参加不必要的会议，在参加会议之前先了解会议的内容，最好能在参会前大致了解各个参会者的想法，尽可能在开会前有自己的大致想法。在参会过程中，一旦发现会议变成了注定无结果的讨论，就及时提出先结束会议，等大家回去整理好思路后再进行讨论，或者线下沟通有争议的内容。在敏捷开发中提倡开站立式会议，通过这种形式的会议，可以有效缩短开会时间。

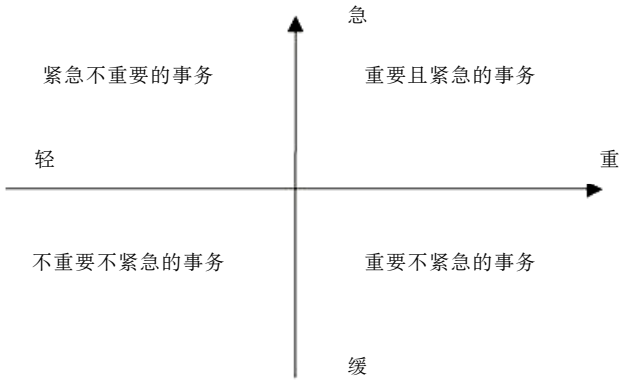
（2）关于讨论：尽量不要让不紧急的讨论打断正在做的事。很多时候，我们正在写代码，这是一个需要大块时间并且思路流畅的过程，一旦被打断，就需要从头开始思考。在日常工作中，有些讨论是不紧急的，那就先告诉要找自己讨论的人，自己有更紧急的事情在做，等自己做完之后再进行讨论。

（3）关于答疑和工作沟通：可以提前与合作方约好，提前告诉他们自己有哪些时间是需要专注地写代码的，让他们尽量不要在这个时间段打扰自己。

总之，要主动管理起自己的时间，不要让自己处于被动中。

2. 设置事情的优先级

美国的管理学家科维提出一个时间管理理论，把工作按照重要和紧急程度进行划分，基本上可以分为四个“象限”：紧急且重要、重要但不紧急、紧急但不重要、不紧急也不重要，如图 2.1 所示。



时间管理坐标体系

图 2.1

对于工作繁杂的程序员来说，这套理论非常实用，可根据这套理论将手头的事情分一个优先级。

◎ 对于重要且紧急的事情，把它们安排在第一时间处理。
◎ 对于重要但不紧急的事情，要避免它们变成重要且紧急的事情，在处理重要且紧急的事情之余，要把重要但不紧急的事情进行拆解，并制定计划，按部就班地完成。
◎ 对于紧急但是不重要的事情，首先想办法看看能不能把它们变成不重要也不紧急的事情，如果还是无法改变，则可以考虑和同事分担。
◎ 对于不重要也不紧急的事情，不要做！

3. 说不

在工作中，我们有时会遇到一些真的需要说不的事情。比如产品经理在需求不明确的情况下找我们做需求评审；项目开发到一半，产品经理突然提出一个不合理的需求变更；本该属于其他领域的事情，其他人为了不想做改造，需要使用你的系统进行改造和兼容等。遇到这些情况，要坚持说不。

当然，说不并不是为了节省时间毫无理由地拒绝。有一句话是这样的：对于需求，先通过讨论解决，解决不了再用代码解决。这才是正确的顺序。这不是指程序员要尽最大努力减需求，而是要对不合理的需求说不。

4. 想清楚再做

笔者刚刚参加工作时，每接到一个需求，都是先动起手来，然后边写代码边思考。刚开始，对于小需求都能应付得过来，但是随着新的需求越来越重要，发现这样处理需求会浪费很多时间，因为经常会发现之前的代码有问题，需要推翻重写。后来笔者对每个很小的需求都会写一份设计文档，在彻底想好如何做之后再开始写代码。这么做便可以大大地节省时间。

在阿里巴巴，技术项目都是需要做设计评审的，这一方面是让合作方了解上下游之间的依赖关系，更重要的是让开发者彻底想清楚所有流程，这样才能在开发时按照自己的设计按部就班地实现。没有想清楚就动手写代码是会带来很多麻烦的，尤其是现在很多公司都在实施服务，各个应用之间依赖关系复杂，我们经常要给外部团队提供接口，如果在定义这个接口时没有想清楚，在上线后发现定义得不完美，就需要合作方配合修改，这样的成本要高很多。

其实，不管是做业务项目还是做技术改造，甚至对于小小的代码优化，都应该先思考一遍，尽量把涉及的细节都想清楚，在确认没问题之后再动手，这样可以大大节省后面的时间。

2.1.7　打破边界

张一鸣曾经公开谈到自己做事从不设边界。他在负责技术时，若遇到产品上有问题，也会积极地参与讨论并构想产品方案。他说：“你的责任心，以及你希望把事情做好的动力，都会驱动你做更多的事情，让你得到更大的锻炼。”

不设边界，看起来吃亏且多做了事情，但成长最快的是自己。

2.1.8　写业务代码中的成长机会

写业务代码有成长机会吗？关于这个问题，答案非常肯定：必须有成长机会。对于大部分公司而言，能够写底层代码或者中间件代码的人总是有限的，写业务代码会面临更高的复杂度。这里分三个层次来看其中的成长机会。

◎ 第 1 个层次，让代码写得不一样。可从代码规范、可读性、可扩展性等角度着手，这也是程序员的基本功。

◎ 第 2 个层次，考虑业务问题和技术问题的匹配。可从写业务代码中理解需求，并做好分析与设计。被动接收需求和实现接口，确实成长空间不大。
◎ 第 3 个层次，总结相关方法体系，成为业务及技术双料专家。

下面通过例子分别针对这 3 个层次进行讲解。

1. 让代码写得不一样

这里看一个例子，常见的代码如下：

```
 if (width <= 1300) {
    $body.addClass('main1200');
} else {
    $body.removeClass('main1200');
}

if (width >= 1580) {
    $body.addClass('screen-xl');
} else {
    $body.removeClass('screen-xl');
}

if (width >= 1400 && width < 1580) {
    $body.addClass('screen-l');
} else {
    $body.removeClass('screen-l');
}

if (width >= 1180 && width < 1400) {
    $body.addClass('screen-m');
} else {
    $body.removeClass('screen-m');
}

if (width >= 940 && width < 1180) {
    $body.addClass('screen-s');
} else {
    $body.removeClass('screen-s');
}
```

这段代码其实可以写成这样：

```
    const $body = 'body';
const bodyWidth = 1000;

// 屏幕尺寸范围
const units = [
{ min: 940, max: 1180, className: 'screen-s' },
{ min: 1180, max: 1400, className: 'screen-m' },
{ min: 1400, max: 1580, className: 'screen-l' },
{ min: 1580, className: 'screen-xl' }];

/**
* $body class 赋值
* @param {Number} width
*/
const assignClass = (width) => {
for (let i = 0, n = units.length; i < n; i++) {
const item = units[i];
const { min, max, className } = item;
if ((!min || width >= min) && (!max || width < max)) {
$body.addClass(className);
} else {
$body.removeClass(className);
}
}
};

if (bodyWidth <= 1300) {
$body.addClass('main1280');
} else {
$body.removeClass('main1280');
}

assignClass(bodyWidth);
```

2. 考虑业务问题和技术问题的匹配

这里举一个烟囱型架构重构的例子。某团队在早期发展阶段对新来的每个业务都会搭建一套系统，业内人士将这种见招拆招、垂直化发展的架构称为“烟囱型架构”。烟囱型架构并非一无是处，在早期业务成败未知的情况下，不过度设计架构，能直接、有效地支持业务。

不过，随着业务的发展，“烟囱”越来越多，对这些“烟囱”的后续维护成为很大的问题，“成长”的烦恼如期而至，如图 2.2 所示。

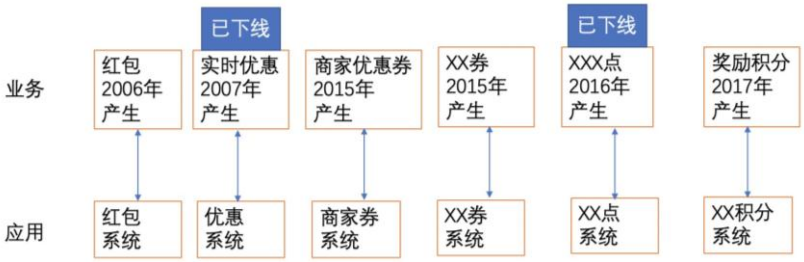


图 2.2

其中存在的问题如下。

◎ 人手不够，业务响应慢了下来。以一个 5 人研发团队为例，起初这个团队从 0 到 1 进行建设，5 个人花了 1 个月做出一个简单版的红包系统；几年后该团队增加到 10 个人，但手头要维护 10 个系统，每个人就要维护一个系统。这时又来了两个新业务，该团队派出 3 个人去做这两个新业务，大约要花 4 个月才能完成。这严重不符合前端业务的响应预期。

◎ 重复建设。在同类烟囱系统中有 80%的功能是类似的，从数据库模型到主要业务逻辑都是复制粘贴加补丁，一不留神就又踩到一个坑。

◎ 维护成本高。面对日常升级包、咨询支持服务，该团队疲惫不堪。

那么，能不能将其中 80%甚至 90%的共性问题抽象出来呢？核心领域模型是否可以稳定呢？如图 2.3 所示，这是可以做到的。

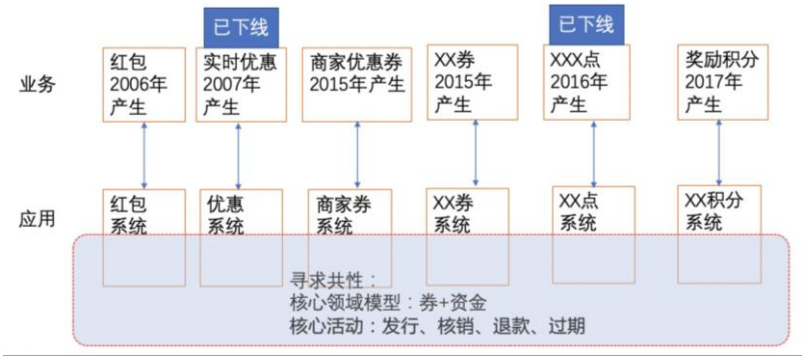


图 2.3

在既要支持不断出现的各种业务，又要建设新平台的纠结中权衡之后，该团队首先启动了平台化项目，建设路径是存量业务继续使用烟囱架构，但新业务随着新平台一起接入，然后逐步迁移存量业务，实现烟囱系统的下线。如图 2.4 所示，优惠券平台对前端业务提供统一的能力输出。

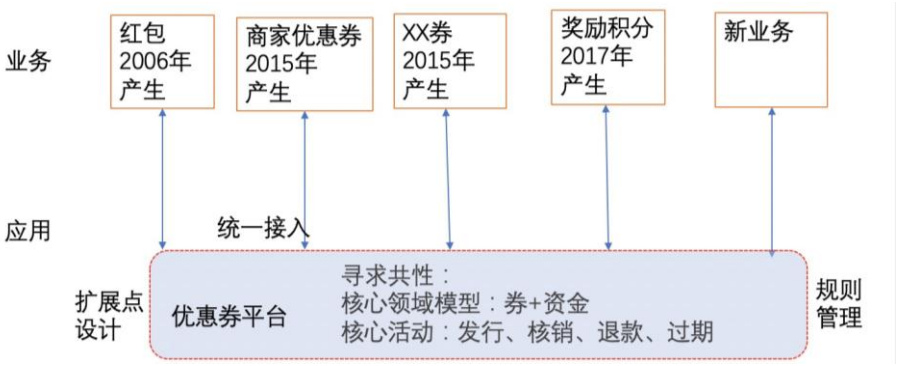


图 2.4

总结下来，平台化架构有如下好处。

（1）快速支撑、响应业务。

（2）抽象共性、边界清晰。快速支撑、响应业务是以终为始的出发点。

架构不服务业务，那么再高大上都是空话。技术不是炫技，要服务于商业。对于抽象共性的问题，业务平台化要解决业务的共性问题，比如天猫、淘宝都有各类营销活动，那就抽象出一个营销平台来管理营销活动、营销工具的整个生命周期，并提供给前端业务使用。

3. 总结相关方法体系，成为业务及技术双料专家

举个例子，曾有一位做事非常努力、成长也比较快的小兄弟诉苦说，他之前在做网关程序，做得很细，除了完成了业务还保障了系统没有 Bug，还使文档沉淀、效率提升，外部机构联调合作方比如银行对其反馈也很好，但大家对其印象是善于合作，技术貌似一般。这位小兄弟在最近一年又做了业务平台，觉得自己在分析设计、中间件技术的应用上已经进步很大，甚至对自己的成长也很满意，但其他人还是认为其技术貌似一般。这位小兄弟百思不得其解。

笔者的反馈如下。

（1）从量变到质变是有一个过程的，自我感觉总是良好的，外部感知可能会晚。如果坚持去做，“迟到的”一定会到。

（2）大部分人都是固定型思维模式和成长型思维模式的混合模式，如果你身边的人固守对你的原有看法，不是你的错，你需要做的就是不断拓展自己的能力。

再举个例子，小郭在几年前参与了一个接入类业务，当时已经有不少机构接入了这个业务，业务规模不大不小，产品经理换了几茬，研发团队也变更了三次。当大家日复一日地做业务接入、测试、发布时，有人发现这个业务缺少一个业务视图。也就是说，这个业务有对系统资源的检测、对接口调用的监控，但是没有对业务运行情况的监控，看不到地区粒度、子业务粒度的变化情况等，甚至 BD 在签订新业务时都不了解为啥某地区已经没有调用量了。小郭利用业余时间开发了一个业务视图工具，整个团队马上就感受到他的过人之处了。有人讲，大公司不是应该什么都就绪吗？实际情况是，大公司也是经历了业务的高速发展的，可以这样说，大部分公司并不缺少做得更好、做得不一样的机会！

所以，“写业务代码有成长机会吗”这个句式还可以改为“维护老系统如何晋升”“做商户接入如何走向高端”和“项目这么忙如何学习”，我们需要进一步将自己的知识和经验系统化，形成相关的方法体系。

在心态上，笔者提两点建议：一是欣赏自己的成长，二是做个有心人。

2.2 学会学习

前面提到过，在这个技术高速发展的时代，做一个不被抛弃的程序员其实是挺难的，因为几乎每天都会有新的技术诞生。

2017 年 8 月，JCP 执行委员会提出将 Java 的发布频率改为每 6 个月一次，并且该决定将在 Java 9 正式发布之后开始实行，新的发布周期严格遵循时间点，将在每年的 3 月和 9 月发布。2019 年 3 月，Java 12 正式发布，Java 13 也会在 2019 年 9 月发布，如果你是一名 Java 开发者，那么这也意味着你所使用的开发语言每半年就会有一个新的版本迭代出来。

这是一件好事，也是一件坏事：Java 开发者可以快速地使用新版本的 Java 语言，更快地体验新的特性，但如果不能主动且高效地学习，那么 Java 开发者被淘汰的速度会比

以前更快。

大多数开发者都深刻理解学习新技术的重要性，但是学习新技术的秘诀是什么？如何快速、高效地学习新的框架、类库和新技术呢？本节将介绍高效学习的几种方法和途径。

2.2.1　高效学习的方法

说到学习，大家可能并不陌生，我们从小到大都经历过很长一段时间的学习，但同样是学习，有些人就能做到很高效，有些人却会浪费大量的时间。不可否认的一个重要原因是，那些高效的人掌握着一套高效学习的方法。

笔者认为，要想比其他人更加高效地学习，就只需做到如下三点：管理好自己的目标、利用好碎片时间，在同一时间只做一件事。

1. 管理好自己的目标

有一个明确的目标是做成一件事的基础，在学习上也一样。很多人整天都在忙忙碌碌，也将很多时间用于学习，但是收效甚微，归根结底就是不会管理目标。要想高效地学习，则不仅仅是制定目标这么简单。很多人都很擅长给自己制定目标，但基本上达不到目标，这就是只会制定目标而不会管理目标的结果。

目标管理，主要可以分为如下几步：评估能力、制定目标、评估目标。

1）评估能力

目标管理的第 1 步，就是正确评估自己的能力。每个人都有不同的能力，并且自己其实是最了解自己能力的人。在评估自己的能力时，可以给自己画一个简单的雷达图，并正确评估自己在各个方向上的能力指标。例如在学习方面，可能要依赖于如下几种能力：理解能力、自控能力、坚持能力、智商能力和专注能力，如图 2.5 所示。

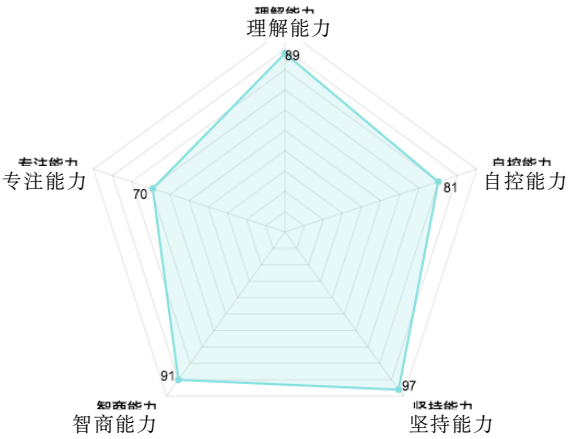


图 2.5

2）制定目标

在正确评估自己的各方面能力之后，就可以根据实际情况及时间等因素，给自己制定目标了。在制定目标时，建议给自己制定两个目标，一个长期目标和一个短期目标。短期目标可以是由长期目标分解而来的。比如计划在三个月内学会 Java 9 的新特性，那么可以计划第 1 个月学会 Jigsaw，第 2 个月学会 JShell，第 3 个月学会 Stack-Walking API。

这里的目标其实等同于计划，不仅有自己要达成的目的，还伴随着时间节点。目标的设定模板可以是：在什么时间节点完成什么事。

3）评估目标

在评估好能力并指定好目标之后，还要养成定期评估自己目标的习惯。这就是我们为什么要设置短期目标，以及为什么要有时间节点，因为这样可以方便我们进行目标的回顾和调整。例如，还是对于学习 Java 9 的新特性这件事，在第 1 个月要完成时，要及时检查一下自己是否真的学会了 Jigsaw，并根据实际完成的情况重新评估自己的能力和制定新的短期目标，以期完成自己的长期目标。

2. 利用好碎片时间

开发者很少有一整块时间可以用来学习，因为平常可能需要加班到很晚，到了周末还要陪陪家人。但是，没有整块的时间不代表没办法学习，如果利用好了碎片时间，则也可以很好地学习。

哪些是碎片时间呢？早晚的通勤路上、午饭后午休前的时间、周末陪孩子上辅导课的时间、甚至是蹲在厕所里的时间，这些都是碎片时间。其实，平时刷朋友圈、刷微博、刷抖音的时间，都是可以用来学习的碎片时间。

在碎片时间里，不适合掏出一本书来看，但可以通过手机或者 Kindle 等学习。现在，很多技术知识都可以通过网上文章、有声书籍、在线视频的方式获取，只要你把刷朋友圈改为阅读技术文章，将刷微博改为听有声书，将刷抖音改为看在线课程，就可以很好地利用碎片时间。

3. 在同一时间只做一件事

前面介绍过高效学习的方法，一个是管理好目标，另外一个是利用好碎片时间。那么，在碎片时间里如何更好地达成目标呢？那就是专注，即在同一时间只做一件事。在学习方面就是指在同一时间只专注地学习一种技术。

比如计划学习 Java 9 和 Spring Boot 2.0，则最糟糕的做法是同时或者交叉学习，这会导致两个技术都学不好。只有专注，才能达到高效学习的效果。

通过管理好自己的目标、利用好碎片时间及在同一时间只做一件事这三种方法，读者就可以高效地学习了。

2.2.2　高效学习的途径

接下来简单介绍开发者学习新技术的途径。

1. 阅读书籍

开卷有益，一般来说，学习知识最好的途径就是阅读，这是一种很古老也极其有效的方法。我们正处于一个信息爆炸的时代，经常刷朋友圈、新闻、微博及知乎等，这些平台分别为我们提供了不同领域的内容，但这些内容大多数比较零散，不成体系。阅读书籍则是一种比较好的选择，而且对于技术类知识来说，书籍中的内容也相对权威一些。

如果你想高效地学习，那么阅读相关书籍绝对是很好的选择，它可以让你对所学的知识建立一个很好的知识体系，并且在学习过程中通过反复翻阅来巩固那些容易遗忘的知识点。另外，你在读书的过程中有足够的时间来思考，这个思考的过程在学习中是十分关键的。

最近一年来，国内的很多听书软件也渐渐火起来了。这类软件的好处就是会把别人写下来的内容念给读者听，有的将总结念给读者听，但听众在听书过程中可能没有太多的思考过程，而且，听过的内容容易忘记，知识点也不便于检索。

2. 参加在线课程

对技术知识的学习，主要包含两方面：原理和实战。对于原理性的技术知识，建议通过阅读书籍等方式来学习；对于实战性的技术知识，则可以考虑学习在线课程，这或许会更高效一些。

在互联网技术方面，有很多在线课程，既包括直播也包括录播，这种学习方式比较直观，可以看到真正的操作步骤，所以很适合实战类的知识学习。比如在学习 Spring Boot 应用开发及部署时，有一小段视频讲解的话就会比较容易理解。

但是在线课程也有一定的弊端，比如不容易使参与者集中注意力，没有办法带动参与者一起思考等。所以，笔者认为，这种视频课程比较适用于实战类的知识学习。

3. 参与线下会议

线下会议，也是近年来比较流行的一种方式，适合有一定经验的开发者，人们可以在会议中和其他人进行交流。很多时候，别人的经验是很宝贵的，所以，这也是一种很不错的学习方式。国内也有很多类似的线下会议可供参加，比如 QCon、ArchSummit 等。

4. 贡献开源项目

对于有经验的程序员来说，保持自我进步的最好方法就是参与开源项目。例如，如果你需要学习像 Apache Spark 或 Hadoop 这样的大数据框架，那么也可以作为开源贡献者参与这个项目。

开源项目的参与者不仅可以根据自身经验来帮助开源社区成长，也可以很好地学习这种新技术。一般的开源项目都是由很多优秀的程序员共同协作的，在参与的过程中和优秀的程序员交流也是一种很好的增长经验的方式。

比如，如果最近想学习 Docker，那么完全可以到 GitHub 上将 Docker 的代码 fork 过来，然后 clone 到本地进行学习。在学习过程中，如果发现问题或者有更好的改进方法，则可以通过 pull request 的方式向原项目贡献自己的代码。而且，这些热门的新技术都会有自己的开源社区，在社区中和其他开发者交流，也是很好的学习过程。

5. 编程竞赛

解决问题其实是提升开发者编程能力的最好方式。除了在日常工作中遇到的一些问题，我们还可以通过参加一些编程竞赛来挑战自己，通过在竞赛中解决编程难题，来快速地学习编程知识和提升编程技能。

目前，国内外有很多编程竞赛网站可供我们选择，例如 TopCoder、CodeForces、Hackerrank、LeetCode 等。

6. 创建博客

没错，创建并记录博客也是一种学习技术的好方式。“教是最好的学习”，学习某技术的最终效果，就是能够把学到的新知识经过自己的理解给其他人讲清楚。相比于其他几种学习方式，创建博客也有很多好处。

◎ 能更深入地了解技术。在很多时候，当我们想学习一个知识并且把它写成文章时，为了避免文章的内容出错，一般都会尽量阅读很多文献，来确保内容是正确的。所以，写文章之前的准备也是自我学习的一个阶段，而且学习的内容往往是比较深入和透彻的，这样才能保证自己写的内容具备权威性。

◎ 能更好地记忆知识点。写作的过程就是最好的记忆过程。因为在写文章的过程中会仔细推敲每一句话。这个过程就是多次记忆的过程。很多时候，你记不住别人说的话是因为其思维方式和你的不一样，但是通过自己的思维方式表达出来的内容就会很容易记住。

当读者试着写一篇关于 Java 中 synchronized 的实现方式的文章时，也许首先会去研究 javac 的编译及 javap 的反编译，通过反编译后的代码了解到 synchronized 是通过 ACC_SYNCHRONIZED、monitorenter 和 monitorexit 实现的，然后会深入 HotSpot 源码并了解其背后的锁优化等技术。写好一篇文章，背后可能涉及十几个知识点，这无疑是一种非常好的学习方式。

7. 编码训练营

“编码训练营”听起来像是一个新鲜事物，但其实我们一般称其为参加培训机构。这种方式更加适合没有经验的人。很多人没有编程经验，所以可以通过这种方式更容易地过渡到职业生涯中，成为一名开发者。

和其他几种学习方式比较，参加编码训练营需要花费更多的资金，但同时有一些好处，

比如有对应的讲师可以直接辅导自己，可以直接解决问题，有些训练营还提供项目实战等。这种方式更适合非计算机专业的新手。

以上几种高效学习途径，可能适用于不同的程序员。有些程序员发现从书中学习很容易，有些程序员觉得学习网上的课程更好，有些程序员喜欢通过创建真实的项目进行学习。简而言之，选择适合自己的学习方法即可。

2.2.3 舒适区与小步子原则

我们习惯待在舒适区是很自然的事情，有励志书籍就讲了，要跳出舒适区，就要挑战自我。舒适不好吗？有知乎上有不少讨论，其中一个回答比较实在。

我们为什么想跳出舒服圈？因为我们想成长，成长的动机来源于自己的追求和社会的期待。例如，我想跳出总是宅在家里的舒服圈，是因为我想增强自己适应社会的能力（自己的追求），或者也可能是你的爸爸妈妈希望你提高交际能力（社会期待）。

这里着重聊一下学习和跳出为什么是困难的，一是动力是否来自自己；二是担心失败的焦虑和负面反馈。比如有人在朋友圈发减肥健身的照片，就可能有人在调侃 A8 腰不是这么减的！因此要做调整，就应该小频度尝试（小步子原则）、快速反馈（及时反馈原则），通过迭代拿到正面的反馈，逐步消除对挫折的潜在畏惧。编程语言几乎开头都是“Hello World”，可能就是出于这样的考虑。

2.2.4 两个著名的学习理论

下面看看两个著名的学习理论。

1. 德雷福斯模型

德雷福斯模型是一种衡量人们的工作方法和能力，反省并提高专业技能的层级模型。它把从新手到专家的技能阶段分成五个层级，如图 2.6 所示。

- ◎ 新手（Novice）：新手需要指令清单，在该技能领域经验很少或者没有经验。这里提到的经验，指的是通过实施这项技术促进了思维的改变。
- ◎ 高级新手（Advanced Beginner）：高级新手不想要全局思维。一旦经过新手的历练，人们就开始以高级新手的角度看待问题。高级新手能够或多或少地摆脱固定的规

则，可以独自尝试任务，但仍难以解决问题。

◎ 胜任者（Competent）：胜任者能够解决问题，处于这一水平的人通常被认为“有主动性”和“足智多谋”。他们往往在团队中发挥领导作用（无论是否有正式的头衔），既可以指导新手，也不会经常“骚扰”专家。

◎ 精通者（Proficient）：精通者能够自我纠正。精通水平的从业者需要全局思维，将围绕某种技术，寻找并想了解更大的概念框架。对于过于简单化的信息，他们会非常沮丧。

◎ 专家（Expert）：专家凭直觉工作。专家是各个领域知识和信息的主要来源，总是不断地寻找更好的方法和方式去做事。他们有丰富的经验，可以在恰当的情境中选取和应用这些经验。他们写文章，并做巡回演讲。

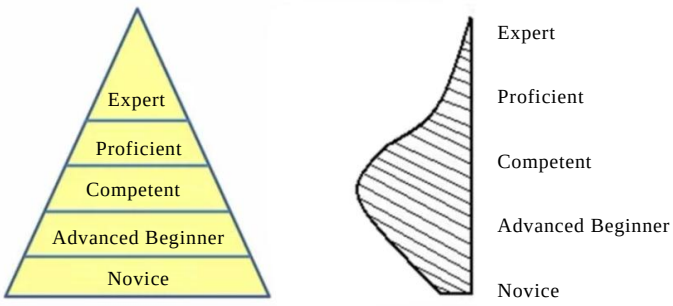


图 2.6

《程序员的思维修炼》的作者针对德雷福斯模型给出一些诀窍。

◎ 新手使用规则；专家使用直觉。

◎ 新手通过模仿和观察来学习；专家要保持实践来维持专家水平。

2. 刻意锻炼理论

首次提出“刻意练习（Deliberate Practice）”这个概念的是佛罗里达州立大学的心理学家 K. Anders Ericsson。这套练习方法的核心假设是，专家级水平是逐渐练出来的，而有效进步的关键在于找到一系列的小任务让受训者按顺序完成。这些小任务必须是受训者正好不会做，但又正好可以学习和掌握的。完成这种练习要求受训者思想高度集中，这就与那些例行公事或者带娱乐色彩的练习完全不同。

2.2.5 终身成长

在经过数年研究之后，斯坦福大学心理学家卡罗尔·德韦克发现了思维模式的力量。她在《终身成长》中表明：我们获得的成功并不是能力和天赋决定的，更受到我们在追求目标的过程中展现的思维模式的影响。

终生成长理念将人的思维分为固定型思维和成长型思维。

◎ 拥有固定型思维的人喜欢与人对比，并不断证明自己最棒，挫折、批评和否定能让其收到打击或者重创。
◎ 拥有成长型思维的人则只考虑能不能够学到知识，可不可以变得更强，能不能够继续成长。

其实，我们耳熟能详的名人也不是一开始就这么强的，比如乔丹。在高中时，乔丹被校队淘汰了。他沮丧地回家向母亲倾诉，他的母亲说，那就回校好好训练。从此以后，乔丹每天早上 6 点离开家到校练习，并在上大学时进一步训练防守、控球和投篮能力。如果输了比赛，他会在赛后更加努力地练习，即使赢了比赛，也会对不满意的地方寻求改进。所以我们要思考一个问题，乔丹靠的是天赋还是不断努力呢？

有趣的是，在大部分活动比如工作中，我们会在面试环节、晋升环节评价一个人的潜力。潜力是面试官的一种判断，支撑的素材来自面试者叙述的现实案例。潜力是变化的，因为这个面试者也是可以成长的，只要他拥有成长型心态并努力。

第 3 章

业务分析与设计

业务分析指应用特定的方式或方法，把复杂的需求拆解成简单且容易理解的对象，并找出这些对象之间的关系。业务分析也是系统开发中最重要、最困难的阶段，只有依据业务分析的结果，运用合理的思想和方法，才能设计出理想的系统架构。如图 3.1 所示，业务分析与设计是程序员进阶时要具备的最重要的能力，是从产品需求到编码实现的重要手段。

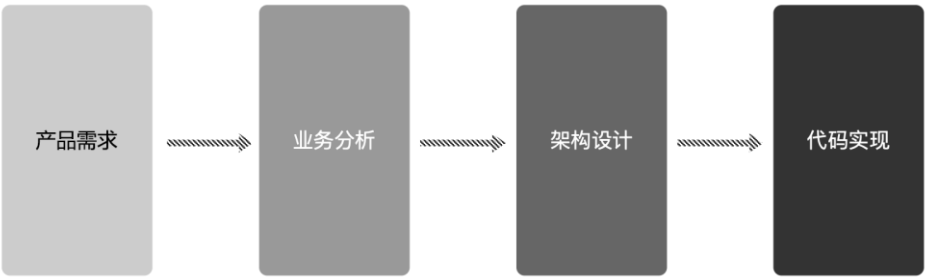


图 3.1

3.1 黄金圈法则

每个人都知道自己要做什么，一部分人知道自己要怎么做，但只有极少一部分人知道自己为什么要这么做，这就是很著名的黄金圈法则，也叫作 2W1H 分析法。黄金圈法则源于知名广告人 Simon Sinek 的发现，是由为什么（Why）、怎么做（How）、做什么（What）三部分组成的由内到外的思维方式，如图 3.2 所示。

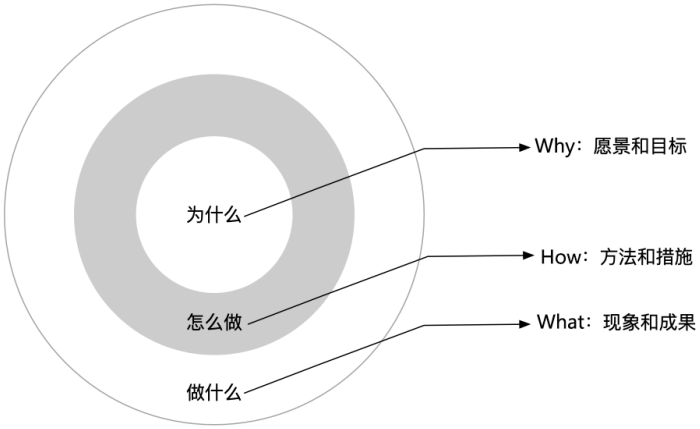


图 3.2

黄金圈法则的三个层次与人脑的三个皮层精确对应，如图 3.3 所示。人脑最外层为新皮质，负责理性思维分析和语言；中间层和最里层为边脑，负责情绪和行动，以及行为和决策。从外到内沟通时，通过说明“做什么”能够帮助我们理解大量的复杂信息，利用的是理性思维，它擅长分析，但不会促使人采取行动。但从内到外沟通时，是直接对应控制决策过程的。



图 3.3

如果将黄金圈法则对应于工作的话，则大部分人都知道做什么，能使用技术将工作做好，通常处于执行层；有些人知道怎么做，具备组织、管理和协调能力，通常处于管理层；

只有极少一部分人知道为什么做，能够制定战略、目标和计划，具有宏观把控能力，通常处于决策层。企业中的执行层、管理层和决策层就形成了人才金字塔，如图 3.4 所示。

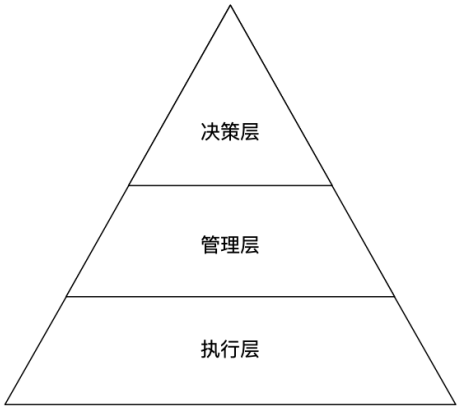


图 3.4

所以，我们通过黄金圈法则能够解释为什么一些组织者或领导者能够在别人觉得不可能的地方激发出灵感和潜力，表现出卓越的天赋。那么，程序员如何从被动接收任务的执行层，进阶为具备系统分析和架构设计能力的决策层呢？有什么方法或捷径吗？答案之一是对 UML 建模工具的熟练运用，因为程序员除了需要具备写代码的能力，还需要具备凌驾于编程语言之上的能力，即系统分析与设计的能力，UML 就是用于系统分析与设计的一种工具。

3.2　UML 建模工具

UML 是一种统一建模语言，是面向对象系统开发过程中非常重要的一部分，主要使用图形符号来表示软件系统的设计，可以帮助项目团队进行内部沟通，找出潜在的需求点并进行设计和验证。

如图 3.5 所示，UML 图可分为用例视图、设计视图、进程视图、实现视图和拓扑视图。用例视图主要有用例图；设计视图主要有类图和对象图；进程视图主要有状态图、活动图、序列图和协作图；实现视图主要有构件图；拓扑视图主要有部署图。

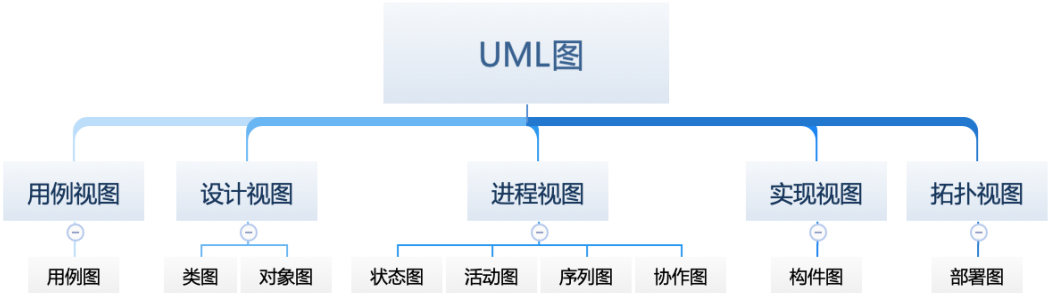


图 3.5

3.2.1 用例图

用例图（UseCase Diagram）是业务分析的产物，描述了系统的参与者与系统的交互功能，是参与者所能观察和使用到的系统功能的模型图，主要用于帮助开发团队以一种可视化的方式理解系统的功能需求，包括基于基本流程的角色关系及系统各个功能之间的关系。它通过用例来捕获系统的需求，再结合参与者（Actor）进行系统功能需求的分析与设计。用例图的关系有包含（Include）、扩展（Extend）和泛化（Generalization），如图 3.6 所示。

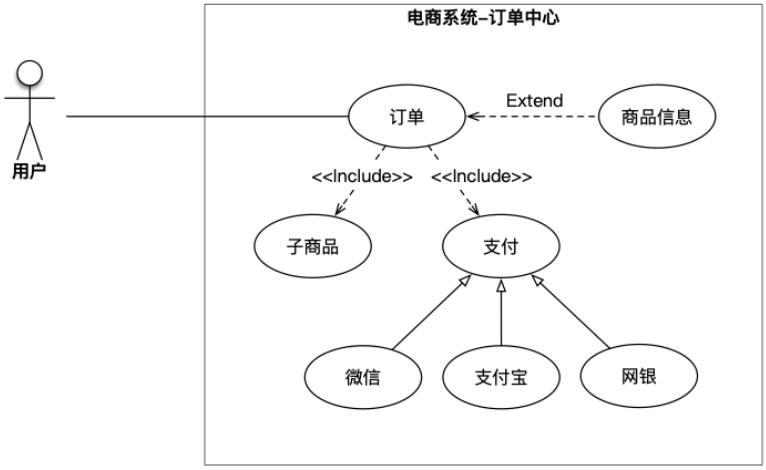


图 3.6

◎ 包含：该用例包含其他用例的行为，并将所包含的用例的行为作为自身行为的一

部分。

◎ 扩展：将某用例的行为纳入已有的用例中，所获得的新用例被称为扩展用例，已有的用例被称为基础用例。

◎ 泛化：子用例将继承父用例的所有结构、行为和关系，而父用例和子用例之间的关系就是泛化关系。

3.2.2　类图

类图（Class Diagram）指用户根据用例图把场景抽象成类，描述类的内部结构，以及类与类之间的关系，是一种静态结构图，如图 3.7 所示。

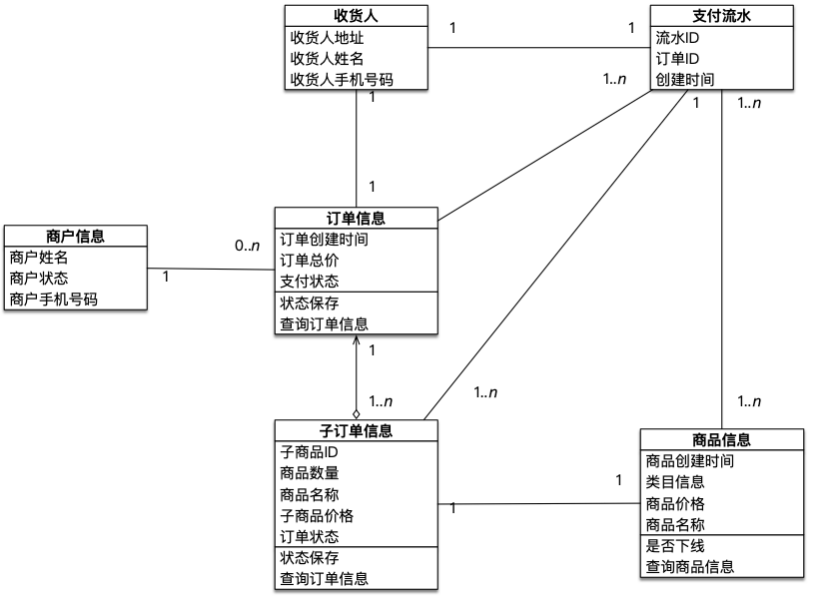


图 3.7

在 UML 类图中，常见的关系有泛化、实现、关联、聚合、组合和依赖，泛化关系由强到弱依次为泛化、实现、组合、聚合、关联和依赖。

◎ 泛化关系：表示类之间是继承关系，为一般与特殊的关系。

◎ 实现关系：表示类与接口的关系，类是接口所有特征和行为的实现。

◎ 关联关系：表示类之间是拥有的关系，它使一个类知道另一个类的属性和方法。

◎ 组合关系：表示类之间是整体与部分的关系。组合关系是关联关系的一种，是比聚合关系还要强的关系，它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期。

◎ 聚合关系：表示类之间是整体与部分的关系，并且部分可以离开整体而独立存在。聚合关系是关联关系的一种，关联关系很强。

◎ 依赖关系：表示类之间是使用的关系，即一个类的实现需要另一个类的协助，所以尽量不要有相互依赖关系。

3.2.3 对象图

对象图（Object Diagram）描述的是一组对象之间的关系，而不是类之间的关系。它是类图的变体，但又与之不同，显示类的多个对象而不是实际的类，如图 3.8 所示。

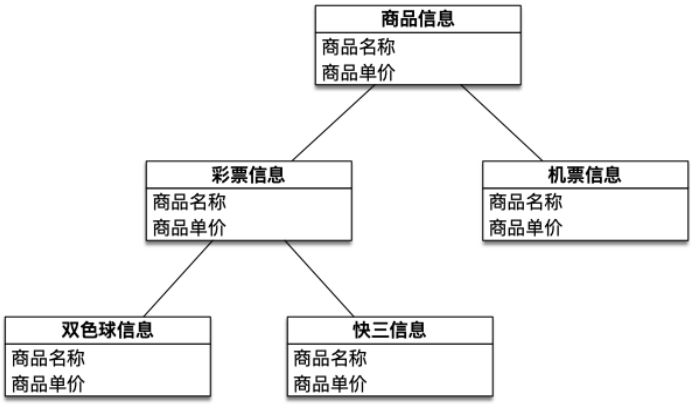


图 3.8

3.2.4 状态图

状态图（Statechart Diagram）用来描述一个特定对象所有可能的状态，以及由各种事件引起的状态之间的转移和变化，如图 3.9 所示。

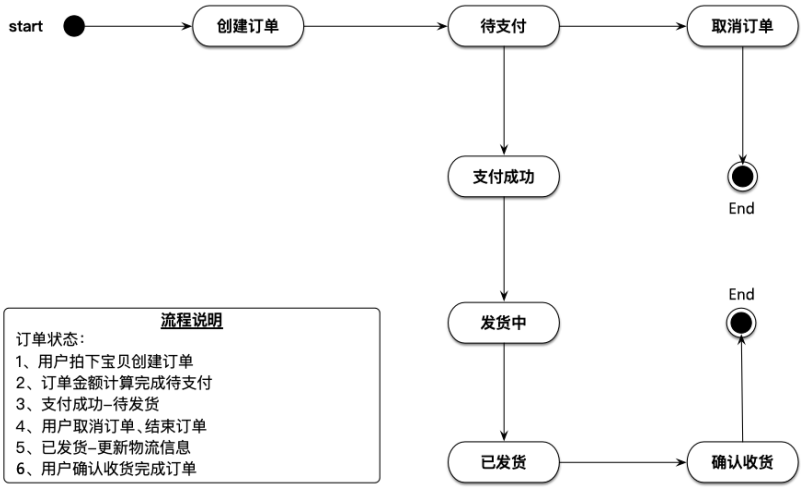


图 3.9

3.2.5　活动图

活动图（Activity Diagram）是状态图的一种特殊图，其中的状态大多处于活动状态。活动图在本质上是一种流程图，与交互图强调从对象到对象的控制流不同，它强调从活动到活动的控制流，如图 3.10 所示。

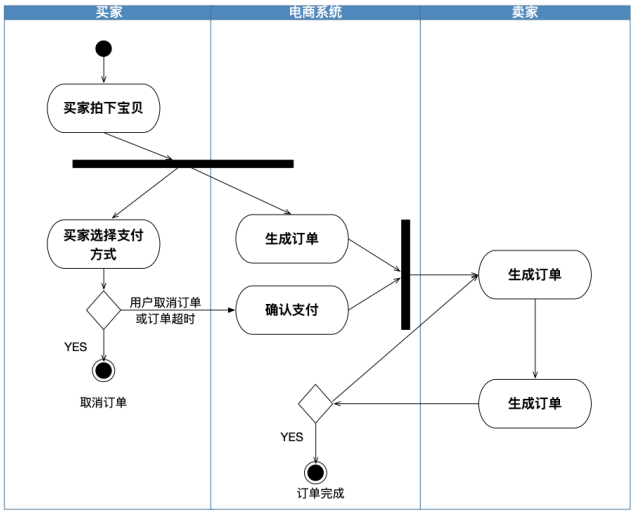


图 3.10

活动图用于表述过程机理、业务过程及工作流程，用来对业务过程、工作流建模，也可以对用例甚至程序实现建模。

3.2.6 序列图

序列图（Sequence Diagram）是交互图的一种，描述了对象之间消息发送的先后顺序，强调时间顺序，主要用于将用例表达的需求做更进一步的精确表达。用例常常被细化为一个或者更多的序列图，序列图能更有效地描述各个类的职责及具备相应职责的原因，如图 3.11 所示。

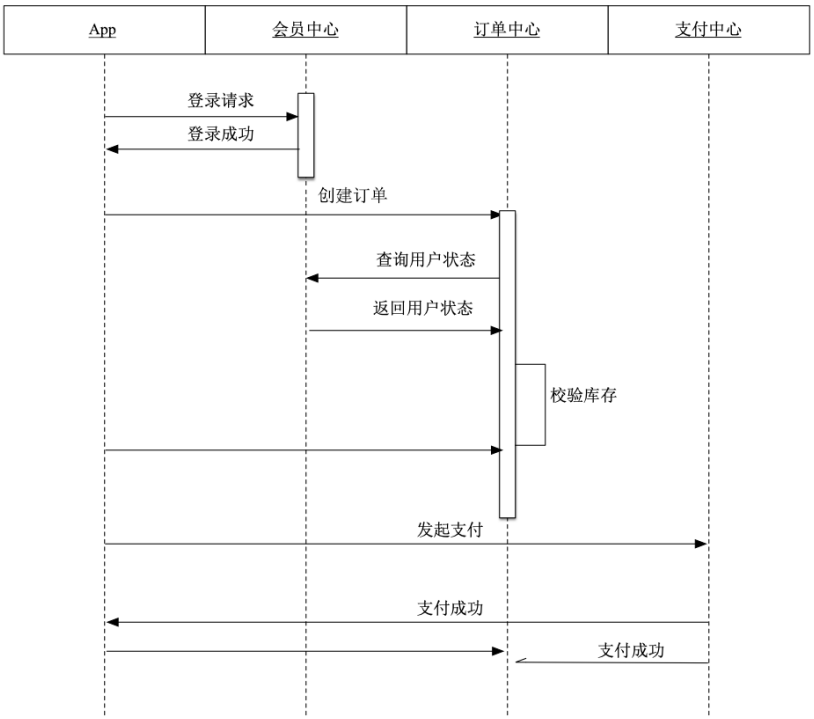


图 3.11

3.2.7 协作图

协作图（Collaboration Diagram）是交互图的一种，描述了收发消息的对象的组织关

系，强调对象之间的合作关系。时序图按照时间顺序布图，协写作图按照空间结构布图，如图 3.12 所示。

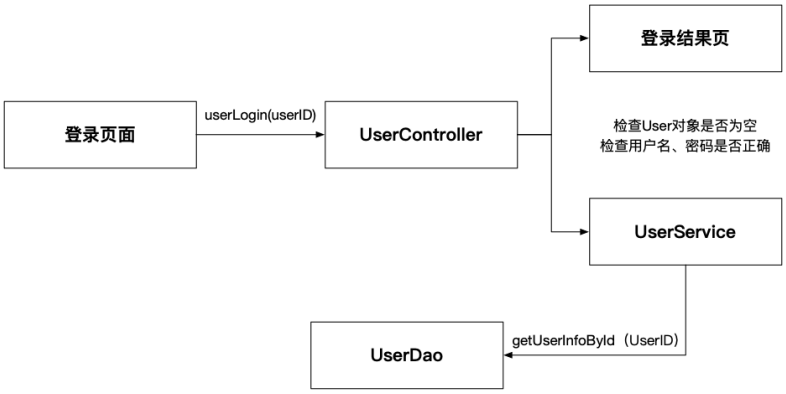


图 3.12

3.2.8　构件图

构件图（Component Diagram）是用来表示系统中构件与构件之间及类或接口与构件之间关系的图。其中，构件图之间的关系表现为依赖关系，定义的类或接口与类之间的关系表现为依赖关系或实现关系，如图 3.13 所示。

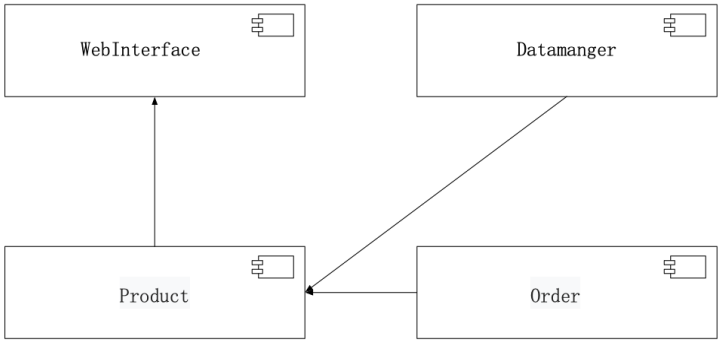


图 3.13

3.2.9 部署图

部署图（Deployment Diagram）用于可视化部署软件的物理组件拓扑，并描述系统的静态部署视图。部署图由节点及其关系组成，强调物理设备及其之间的连接关系，如图 3.14 所示。



图 3.14

在敏捷开发中，每一次迭代都可以被看作一个小型的瀑布模式，通常可以分为 6 个阶段：需求、分析、设计、开发、测试、交付，具体流程如图 3.15 所示。

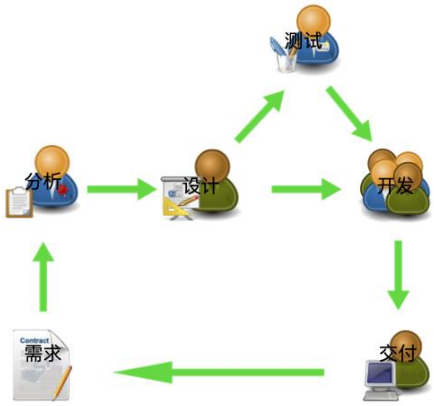


图 3.15

UML 在其中的各个阶段都有不同的应用，如下所述。

（1）需求阶段：采用用例图描述需求。

（2）分析阶段：采用类图描述静态结构，采用顺序图、协作图、活动图和状态图描述动态行为。

（3）设计阶段：采用类图对类的接口进行设计。

（4）开发阶段：采用某种面向对象语言实现。

（5）测试阶段：单元测试采用类图和类的规格说明书；集成测试采用类图、协作图；系统测试采用用例图。

（6）交付阶段：采用构件图和部署图。

3.3　业务分析与设计的方法

现在，用户的需求越来越复杂，变化也越来越快，如果仍然通过需求管理、需求跟踪等管理方式来约束和减少频繁变更需求对软件开发带来的冲击，则会导致软件开发流程变得僵硬，程序员更加疲惫。另外，用户的需求通常来自某个专业领域，比如法律、财务或电子商务等，每个特定的需求又有其特别复杂之处，所以几乎没有人能够第一时间抓住这些专业领域的需求本质。因此，用户的需求在历经几次演变之后变得面目全非，每一个“加工制造环节”都认为自己在做“正确的事”。

在现实工作中，这样的局面一再上演，比如，产品经理在宣讲产品需求文档（Product Requirement Document，PRD）时，需要将专业名词使用通俗的业务语言“翻译”给业务人员，还要使用技术语言“翻译”给开发人员，这就很容易导致各方得到的信息不对等，业务人员的理解和技术人员的理解可能不一致。

因此，聪明的工程师们引入了灵活多变的面向对象分析与设计（OOAD）方法。面向对象分析与设计方法对复杂需求的解决方法是：让建模专家进行需求分析，并在需求和需求之间建立关系。

注意：面向对象分析与设计被拆分为系统分析和系统设计两部分，还专门有“系统分析师”和“系统设计师”两种职称考试。这样拆分的结果就是对需求分析的结果无法直接进行设计和编程，而能够运行的代码不匹配需求，导致客户在运行软件后才发现很多功能不是自己想要的，而且软件不能快速跟上需求的变化。

虽然面向对象分析与设计的思想带给分析与设计很大的灵活性，也对软件开发产生了前所未有的影响，但有一定的局限性：面向对象分析与设计主要关注微观层次的抽象，比如类和对象实例等；对每个问题域通常都需要创建单独的用例分析模型，项目或产品的大方向在许多情况下变得很模糊，很难全局把控系统的总目标，所以这样的系统分析与设计

方式存在很大的风险。

面向对象分析与设计的局限性，催生了面向服务分析与设计（SOAD）方法，它有效地弥补了面向对象分析与设计的局限性。同时，面向服务的架构（Service-Oriented Architecture，SOA）也走进了大众的视野并很快流行起来。

面向服务分析与设计并不是一种全新的分析与设计方式，只不过是从不同的抽象角度去设计一种业务模型。面向对象分析与设计可以说是一种自底向顶的设计方式，虽然面向对象的方法比起面向过程的方法有了很大的改进，但是在面对复杂的市场需求时，面向对象分析与设计就显得捉襟见肘了，此时就需要从更高的层次分解市场需求，面向服务分析与设计应运而生。在做面向服务分析与设计时会采用自顶向底的设计方法，首先要分离出业务流程和服务，然后细化对象。面向对象分析与设计与面向服务分析与设计的抽象关系如图 3.16 所示。

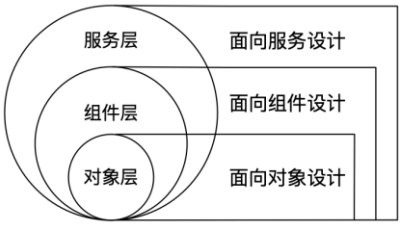


图 3.16

那么，面向服务分析与设计就是最好的了吗？它又有什么局限性呢？

众所周知，软件开发的流程通常为需求、分析、设计、开发、测试和交付。不管是在面向对象分析与设计中还是在面向服务分析与设计中，对系统分析和设计都是分开的、割裂的，即分析人员从领域中收集基本的业务概念，系统设计人员再将基本的业务概念映射为相应的编程工具的构造组件。在这个过程中，由于分析模型和设计模型不同，在分析和设计之间形成一条鸿沟，如图 3.17 所示。



图 3.17

2004 年，著名建模专家 Eric Evans 出版了其最具影响力的著名书籍 *Domain Driven-Design architecture*，其中提出的领域驱动设计（DDD）方法打破了分析和设计割裂的状态，并给出了领域模型的概念，抛弃了将分析与设计分开的做法，使用统一的模型来满足分析与设计的需求，使系统开发能够更加灵活、快速地响应需求的变化。

注意：DevOps 的出现又进一步填平了开发和部署之间的鸿沟，值得注意的是，任何新技术或概念都不是人们凭空想象出来的，都是为了解决人们在生活和工作中遇到的问题。架构师也需要具备发现系统中的问题并给出相应解决方案的能力。

3.4　系统分析与设计的三个发展阶段

下面讲讲系统分析与设计的三个发展阶段。

3.4.1　面向数据驱动分析与设计

面向数据驱动分析与设计可以说是系统分析与设计的第一阶段。软件设计总是从设计数据库及其字段开始的，这一阶段的特征就是围绕数据库编程，应用系统是典型的两层架构，分为展示层和数据库层。该阶段的典型技术为 Delphi 和 VB 等，如图 3.18 所示。

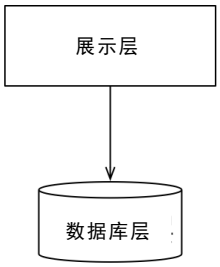


图 3.18

这种面向数据驱动分析与设计的方法导致了过程化的编程思维，因为数据库结构由 DBA 设计后交由程序员编写大量的 SQL 语句实现，而 SQL 语句执行是有先后顺序的，所以程序员大多形成了面向过程的思维方式，长此以往形成了习惯就难以改变。

面向过程（Procedure Oriented）是一种思维方式，在面对一个问题时，我们通常先关注解决这个问题的步骤，即过程。比如对于如何将大象装入冰箱这个问题，我们想到的步

骤是：第 1 步，打开冰箱；第 2 步，装入大象；第 3 步，关上冰箱。这就是典型的面向过程的思维方式，虽然这样可以更加直接、有效地解决问题，但是面对更复杂的问题时，解决问题的过程会变得非常复杂和难以理解。

同样，面向数据驱动分析与设计有以下明显缺点。

◎ 不能迅速、有效、全面地认识和反映需求，在这种分析与设计思维中，世界不仅由简单的关系数据组成，还使用关系数据库反映现实需求，这不符合人类的自然思维，是一种扭曲的分析方法。
◎ 系统的性能很难提升，容易导致软件运行时的负载集中在数据库端，使系统变成集中式和高风险的大型单体模式（Monolithic），丧失分布式集群处理能力。
◎ 面向对象的编程语言和关系型数据库本身是矛盾的，因为关系型数据库分析与设计本身是面向过程的（这一矛盾在当今的实现中依然存在，不过在领域驱动设计中已有解决方案）。

3.4.2 面向对象和服务分析与设计

随着系统的复杂度越来越高，面向数据驱动分析与设计所存在问题也越来越明显，因此产生了具有划时代意义的面向对象和服务分析与设计的方法，应用系统也变成了经典的三层架构：展示层、业务逻辑层和数据访问层，如图 3.19 所示。

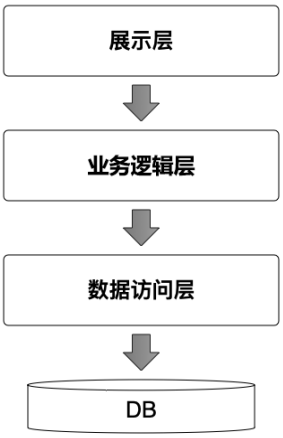


图 3.19

此时出现独立进行分析与设计的两个阶段，系统分析与设计开始上升到一个更高的层次，拥有自己的一套科学且艺术的方法论，但也带来了一个致命的缺点：分析阶段和设计阶段不能很好地衔接，出现了难以逾越的鸿沟，因为分析人员负责从需求领域中收集基本概念，而设计人员负责指明一组能在项目中适应编程工具构造的组件，这些组件必须能够在目标环境下有效执行，并能够正确解决应用程序出现的问题。

可以看到，分析阶段和设计阶段的目标并不一致：分析人员只关注需求分析，并不关注是否适合设计或者能否导出适合设计的分析结果；设计人员发现分析结果过于简单，无法进行编码实现。于是，分析阶段和设计阶段无法对接，导致整个项目无法顺利进行，以失败告终。

另外，在分析阶段和设计阶段虽然都使用了 UML，但是 UML 不是思想方法，只是一种分析与设计工具。假若将 UML 类比为 CAD 绘图软件，那么会 CAD 的绘图员就是建筑师吗？显然不是。所以，UML 不是银弹，更不等价于分析与设计方法。

3.4.3　面向问题域分析与设计

问题域模型有一个流行的名字：领域模型，是对现实世界或领域中的对象的可视化表示，可分为概念模型、领域对象模型和分析对象模型。

Eric Evans 在 2004 年发表了 *Domain-Driven Design –Tackling Complexity in the Heart of Software* 的论文，主题便是领域驱动设计，还提出领域模型的分层架构，将整个系统划分为基础设施层、领域层、应用层和用户接口层，如图 3.20 所示。

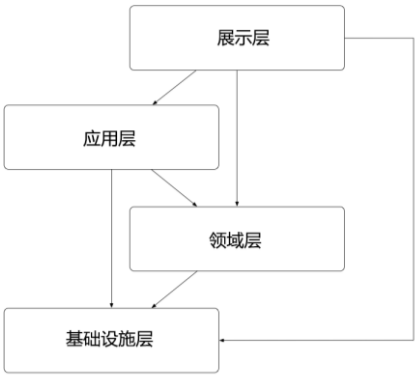


图 3.20

领域建模是一种艺术，融合了分析阶段和设计阶段，目的是使复杂的软件快速应付变化。领域模型同时适用于分析原型和程序设计，如果一个模型在实现时不具备可行性，就需要重新寻找新的模型，如果该模型没有忠实表达领域的关键概念，则也必须重新寻找新的模型。所以，领域建模的过程把分析阶段和设计阶段变成了单个循环阶段，把分析与设计紧密联系起来，使领域建模专家不再只关注需求概念的收集，也关注程序代码的设计与实现。

3.5 ~ 3.7 节将介绍面向对象分析与设计、面向服务分析与设计和领域驱动设计的详细用法。

3.5 面向对象分析与设计

面向对象分析与设计是指根据面向对象方法学对软件系统进行分析与设计。在面向对象分析与设计的定义中有三个关键词：面向对象、分析、设计。所以，为了更好地理解面向对象分析与设计的作用，我们首先要理解什么是面向对象，以及面向对象分析和面向对象设计的原则。

3.5.1 什么是面向对象

面向对象是对现实世界进行理解和抽象的方法，是计算机编程技术发展到一定阶段后产生的一种软件开发方法，具有抽象、封装、继承、多态等特征，还形成设计原则和设计模式的智慧结晶。我们可以使用一张图来更为形象地展现面向对象所涉及内容之间的关系，如图 3.21 所示。

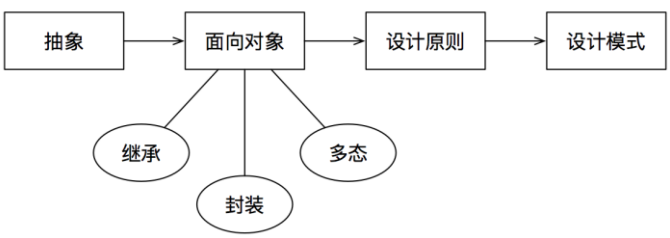


图 3.21

3.5.2　面向对象的特征

面向对象的特征如下。

（1）封装：指将相关属性和方法重组成一个新的对象，并提供对外访问的接口，对数据的访问或修改只能通过该接口进行。电商商品有拍下和支付等行为特征，但其实现细节对用户来说是不可见的，所以在进行商品购买的过程中，用户并不清楚交易是如何进行的，以及订单的状态是如何发生改变的，这就体现了对象的封装性。

（2）继承：指某类对象可以继承其他类对象的属性和方法，是一种类之间的层次关系表达。继承有单继承与多继承两种方式：单继承指子类只有一个父类；多继承指子类有一个以上的父类。继承是代码重用的具体表现之一，也破坏了对象的封装性，父类的公共属性和方法对子类开放，如果修改父类的公共属性和方法，就会导致所有子类的改变，这在一定程度上其实破坏了系统的可扩展性。

（3）多态：指不同的类对象对同一消息做出响应，但同一消息可以根据发送的不同对象进行不同的处理，最终得到不同的执行结果。接口的不同实现方式也被称为多态。

（4）抽象：指为了达到某种目的，对涉及的信息和现象进行过滤，将共性元素进行合并，这些元素同时具有独立性和特殊性。抽象也是面向对象分析与设计的基础。

3.5.3　面向对象设计的原则

为了设计出一个好的软件系统，就必须遵照一定的规则，而衡量软件设计质量的首要标准就是该设计能否满足软件的功能需求。除了功能需求，还有很多衡量软件设计质量的标准，主要包括高内聚、低耦合、可扩展和可复用。

（1）高内聚：系统中的模块具有高度相关的职责，如果除了这些职责内的任务，没有过多的其他任务，该模块就具有高内聚性，反之具有低内聚性。并且系统中的模块与模块之间有较高的协作性，也就是常说的单一责任原则。

（2）低耦合：耦合指元素与元素之间连接、感知和依赖的程度。这里所说的元素，可以指功能、对象（类），也可以指系统、子系统、模块。耦合度越低，系统越独立，可扩展性就会越好，所以在设计系统时首先要考虑对元素的划分。

（3）可扩展：用于应对更大规模的业务及软件的成长，通常采用动态加载的插件、回调函数、抽象接口及认真设计的代码结构和类层次结构，使系统在面对不断变化的需求时，

其代码不被大量重构开发，比如添加新功能或修改完善现有功能。

（4）可复用：又叫作重用，即重复使用，可以利用已有的代码或者相关模块去实现新的功能需求，实现高效、低成本和高质量。

面向对象是一种解决现实问题的思想，让我们把重点放在现实的客体上。我们可以通过 UML 弄清楚这些客体之间的关系，最后把客体与客体之间的关系用面向对象的语言表达出来。客体与客体之间的关系如图 3.22 所示。

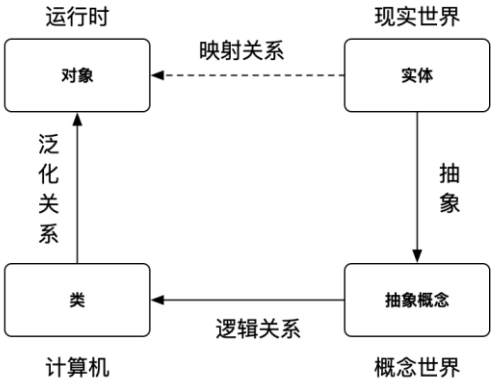


图 3.22

回顾前面介绍的将大象放入冰箱的例子，我们如何使用面向对象思想实现这个例子呢？首先，找出现实世界中的动作和实体：①打开冰箱；②将大象放入冰箱；③关闭冰箱。然后，抽象出概念：①冰箱、可以打开门、可以存储、可以关门；②大象、体重、体积。最后，用计算机中的类来实现该逻辑关系，这样就有了 elephant 和 fridge 两个类。

面向对象的代码实现如下：

```
Class elephant{
private int weight;
private int size;
}
Class fridge{
public void open();
public void store(elephant e);
public void close();
}
```

虽然面向对象起初专指在程序设计中采用封装、继承、多态、抽象等设计方法，然而现在它已经渗透到软件开发的各个方面，比如面向对象的分析（OOA）、面向对象的设计（OOD）和面向对象的编程（OOP）等环节，其在各环节的职责如下。

（1）在需求分析阶段采用面向对象的分析方法，这个阶段不需要思考怎么用程序实现它，只需要思考和分析需求中稳定不变的客体是什么，这些客体之间的关系又是什么，以及完成概要建模。

（2）在设计阶段采用面向对象的设计方法，把在需求分析阶段分析出来的需求进一步扩充为模型，变成可实现的、符合成本的、模块化的、低耦合高内聚的模型。

（3）在编程开发阶段采用面向对象的编程语言来具体实现在前一个阶段得到的业务模型。

3.6　面向服务分析与设计

面向服务（SOA）是一种架构思想，可以是跨语言的，也可以是跨平台的，其宗旨就是根据项目的服务完成架构搭建，以服务为基准点完成系统的组件化和模块化。比如，在电商系统中提供商品搜索服务、商品查询服务、用户信息服务、订单查询服务，等等，这些都是由电商系统提供的服务，这些服务由一个个的业务点组成，一个服务就是一条业务线，整个项目的服务体系可以说是由各种各样的业务点组成的，并由业务线串联而成，而服务的实现过程就是面向服务分析与设计。

3.6.1　Web Service 模式

在 Web Service 模式下，服务提供者创建服务，并决定将其公开并发布。发布是通过将服务信息发布到服务目录上来完成的。另外，服务消费者将在服务目录中搜索满足必要条件的服务，在找到服务后，通过服务目录中的可用服务信息，就能够以正确的方式直接调用服务提供者，从而满足业务的需求。其过程如图 3.23 所示。

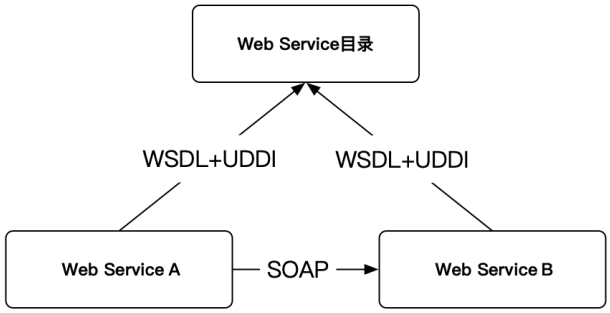


图 3.23

对 Web Service 中的一些术语解释说明如下。

◎ 服务提供者：指服用的被调用方，即为其他服务提供服务的服务。
◎ 服务消费者：指服务的调用方，即依赖其他服务的服务。
◎ 服务目录：用于向消费者提供可以用的服务目录。
◎ SOAP（Simple Object Access Protocol，简单对象访问协议）：是交换数据的一种协议规范，是一种轻量的、简单的、基于 XML 的协议，它被设计成在 Web 上交换结构化的和固定化的信息。
◎ WSDL（Web Service Description Language，Web 服务描述语言）：用来描述 Web 服务发布的 XML 格式。
◎ UDDI（Universal Description Discovery and Integration）：指通用的描述、发现、集成服务，用来存储服务信息的目录及描述服务界面的目录。我们可以通过 UDDI 根据自己的需要动态查找并使用 Web 服务，也可以将自己的 Web 服务动态发布到 UDDI 注册中心，供其他用户使用。

3.6.2 ESB 模式

ESB（Enterprise Service Bus，企业服务总线）提供服务对接、协议转换、消息传输、数据转换、内容路由等功能，隐藏了服务提供者的物理位置、协议和数据格式，由中间件产品技术实现，通过事件驱动和 XML 消息引擎为更复杂的面向服务的架构提供服务。ESB 通常在企业的消息系统上提供一个抽象层，可以帮助设计者利用消息（不用编码）就能完成集成工作，其架构如图 3.24 所示。

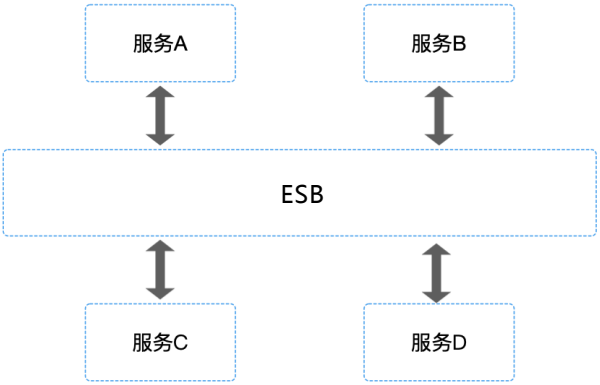


图 3.24

3.6.3　微服务架构

微服务架构是 SOA 架构的延续，是一种比较现代化的细粒度的 SOA 实现方式，强调业务需要彻底组件化和服务化，也就是说，系统需要按照业务边界做细粒度的拆分和部署。这样，原有的单个业务系统会被拆分为多个可以独立开发、设计和运行的微服务系统，这些微服务之间通过 REST（Representational State Transfer）完成交互和集成。

那么，微服务架构又具有什么样的特点呢？可简单归纳为如下三点。

（1）独立部署、灵活扩展。传统的单体架构是以整个系统为单位进行部署的，而微服务架构是以每一个独立组件为单位进行部署的，可以根据每个服务的高并发要求的不同进行不同的部署。比如，服务 A 需要部署 10 台机器，服务 B 需要部署 20 台机器，而服务 C 只需要部署 3 台机器，像这种灵活的部署方案只有微服务架构才能实现，近几年流行的容器化更为微服务架构提供了高效、灵活的部署方案。

（2）资源的有效隔离。微服务设计的原则之一，就是每一个微服务都拥有独立的数据源，假如微服务 A 想要读写微服务 B 的数据库，则只能调用微服务 B 对外暴露的接口来完成，这样就有效避免了服务之间争用数据库和缓存资源所带来的问题。另外，由于每一个微服务实例都运行在不同的容器中，所以更好地实现了服务器资源（比如内存、CPU 等）的隔离，如图 3.25 所示。

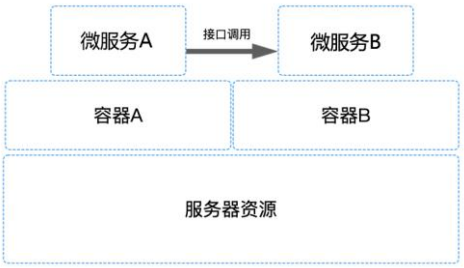


图 3.25

（3）敏捷的组织结构。微服务设计的思想也改变了原有的企业研发团队的组织架构，从以往水平的职能型团队变成了垂直的业务型团队。职能型团队和业务型团队的架构如图 3.26 和图 3.27 所示。

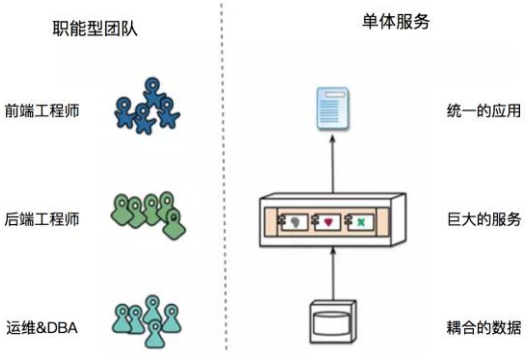


图 3.26

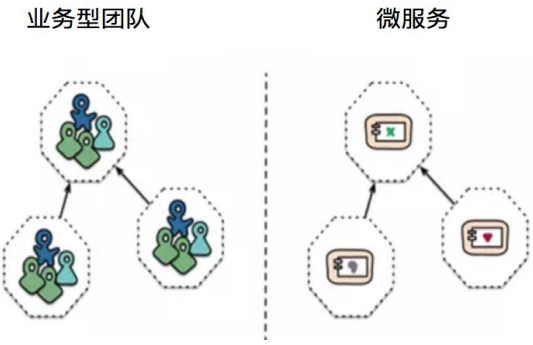


图 3.27

当然，微服务架构也带来了各种各样的问题，如下所述。

◎ 微服务架构把原有的项目拆成多个独立的工程，增加了开发、测试和运维的复杂度。
◎ 微服务架构需要保证不同服务之间的数据一致性，引入了分布式事物和最终一致性机制，为设计和开发带来了挑战。
◎ 很多微服务的拆分粒度过大，导致服务的调用链过长，性能降低，维护成本增加。

领域驱动设计可以弥补微服务架构中的很多不足，为微服务架构提供指导思想，比如服务拆分的粒度和事务合理设计等，下面重点介绍领域驱动设计的详细方法。

3.7　领域驱动设计

领域驱动设计是一种将实现连接到持续进化的模型中来满足复杂需求的软件开发方法，通过将软件的相关部分连接到不断发展的模型中来简化复杂应用程序的创建流程。领域驱动设计侧重于以下三个核心原则。

◎ 把项目的重点放在核心域（Core Domain）和域逻辑上。
◎ 把复杂的设计放在有界域（Bounded Context）的模型上。
◎ 和领域专家不断协作完善应用模型来解决特定领域的问题。

所以，在领域驱动设计中非常重要的一个概念就是领域，而领域又可以划分为问题域和问题解决域。领域中各概念之间的关系如图 3.28 所示。

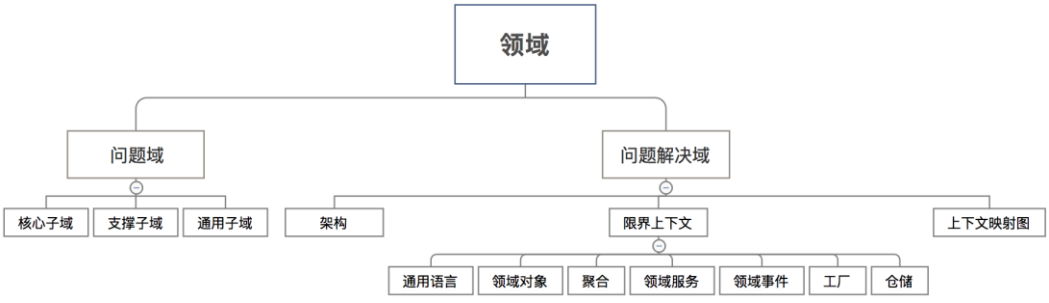


图 3.28

领域指一个组织在某个特定的范围内应用特定的方式进行特定的活动。例如，一个公司通常会先确定一个市场，然后在这个市场上研发产品和提供服务。

领域驱动设计就是先将在领域中涉及的数据、流程、商业规则等都弄明白，然后以面向对象的观点为其建立一个领域模型，再选用合适的软件技术实现这个模型。在领域中涉及的数据、流程、商业规则等就是我们常说的领域问题（问题域），对于大的问题又会按功能或重要程度划分为不同的子问题（子域），采用分而治之的思想；领域模型及使用合适的软件技术去实现个模型就是我们常说的问题解决方案（问题解决域）。图 3.29 展示了领域和各子域的关系。

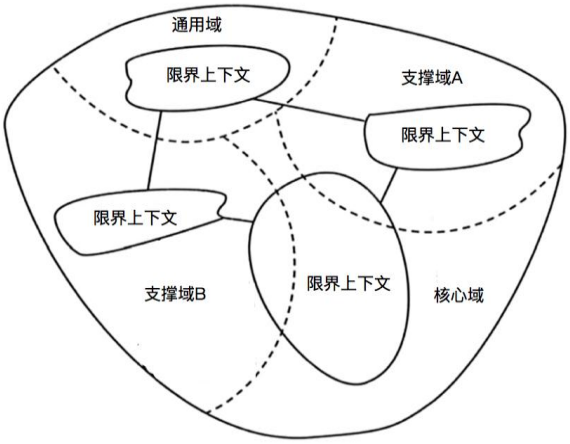


图 3.29

在将限界上下文划分好之后，还要确定限界上下文之间的关系。通过限界上下文的映射关系，能够明确限界上下文之间的耦合关系。这里主要关注限界上下文的内部交互和外部交互的耦合度，以最大限度地减少耦合度的层级。限界上下文之间的关系有以下几种。

◎ 合作关系（Partnership）：两个合作的上下文会相互影响和制约。
◎ 共享内核（Shared Kernel）：两个上下文都依赖部分共享的模型。
◎ 客户方-供应方开发（Customer-Supplier Development）：上下文之间有组织的上下游依赖。
◎ 遵奉者（Conformist）：下游上下文只能盲从上游上下文。
◎ 防腐层（Anti-Corruption Layer）：上下文之间通过适配和转换进行交互。
◎ 开放主机服务（Open Host Service）：指定一套明确的协议，方便其他上下文访问。
◎ 发布语言（Published Language）：在一般情况下和开放主机服务一起使用，用来定义开放主机的协议。
◎ 大泥球（Big Ball of Mud）：混杂在一起的上下文关系，边界不清晰。

◎ 无关系（Separate Way）：两个完全没有联系的上下文。

耦合度就是某模块与其他模块之间关联、感知和依赖的程度，是衡量模块独立性的一个指标，也是系统分析与设计中的重要指标。

接下来详细介绍领域驱动设计中的一些核心要素，如下所述。

（1）分层架构（Layered Architecture）：分层架构的原则之一是只能向下依赖，不能向上依赖。分层架构有严格分层架构和松散分层架构两种：在严格分层架构中，当前层只能依赖其直接下方的层；在松散分层架构中，当前层可以依赖其下方的任意层。领域驱动设计的分层架构图如图 3.20 所示。

（2）应用层（Application Layer）：负责将展示层的请求转发到领域层，将领域层的执行结果返回给展示层，并根据业务对象调用相应的应用服务。它不包含业务逻辑，相对来说是较“薄”的一层。在该层除了可以定义应用服务，还可以进行安全认证、权限校验、持久化事务控制、调用外部系统或者向其他系统发送事件消息等。另外，应用层是展示层与领域层的桥梁，展示层使用视图模型进行界面展示，与应用层通过数据传输对象进行数据交互，从而达到展示层与领域对象解耦的目的。

（3）领域层（Domain Layer）：负责表达业务的概念，实现全部的业务逻辑并且通过各种校验手段保证业务的正确性，是核心部分。业务逻辑包括业务的流程、策略、规则、状态及完整性约束等，所以领域层是较“胖”的一层。

（4）实体（Entity）：实体是一个不由自身属性定义而由它自身身份定义的对象，具有状态和行为两个属性。实体对象具有唯一性和可变性，唯一性由唯一的身份标识来确定，可变性则反映了实体本身的状态和行为。

（5）值对象（Entity Object）：只包含元素属性的不可变对象。值对象用于描述和度量事物，创建、使用、优化、测试和维护起来非常容易。比如对于某个地址（Address）对象，不通过唯一的身份标识 ID 来决定它的唯一性，只通过固定不变的概念来表示一个具体的地址即可。

（6）服务（Service）：服务分为应用服务和领域服务，强调与其他对象的关系，只定义了可以为客户做什么，而不应该替代实体（Entity）和值对象的所有行为，也就是常说的充血模型。

◎ 应用服务：是用来表达用户故事（User Story）和用例的主要手段。应用层通过接口来展示应用服务的功能列表，应用服务负责服务编排和请求转发，同时将需要

实现的功能通过领域对象来实现。应用服务只负责处理业务用例的执行顺序和结果拼装。这样可以很好地隐藏领域层的复杂性和内部实现原理。

◎ 领域服务：表示一种无状态的操作，用来实现某个特定领域的任务。它强调无状态的操作，状态应该在实体中维护，领域服务处理是无状态的逻辑处理过程，实现了某个特定领域的任务，其做的也是领域内的事情，是通用语言的表达。应用服务是领域服务的客户方。当某个操作不适合放在聚合（实体）或值对象上时，最好的方式便是使用领域服务。

（7）模块（Module）：指提供特定功能的相对独立的单元，也就是对功能的分解和组合。我们通常将领域模型分解成不同的模块，来降低领域模型的复杂度，从而提高领域模型的可读性。

（8）聚合（Aggregate）：聚合是由聚合根（Root Entity）绑定在一起的对象的集合，是领域对象的显示分组，用来表达整体的概念（也可以是单一的领域对象）。它的作用是保持领域模型的行为和唯一性，同时作为一致性和事务性的边界。聚合根通过禁止外部对象对其成员的引用来保证在聚合内进行的更改是一致的，所以它的难点一般在对一致性的维护上：在聚合内实现强一致性，在聚合外实现最终一致性。在进行聚合设计时要遵循以下几个原则。

◎ 遵循领域不变性。
◎ 每个事务都只包含一个更新聚合，当在事务中有多个更新聚合时，就要通过领域事件对事务进行拆分，实现最终一致性。
◎ 使用小聚合，避免把聚合当作领域对象的集合或容器，从而出现“巨大”的聚合根。
◎ 不仅仅是 HAS-A 的关系，与面向对象（OO）思想中的聚合不是一个概念。领域驱动设计中的聚合首先要确定对象的行为和唯一性，由此可见聚合不是单纯的包含关系。
◎ 领域对象的持久化和检索是通过聚合根来完成的。
◎ 使用值对象，将聚合根内的其他领域对象优先设计成值对象。
◎ 使用 ID 进行关联，而非对象引用。
◎ 聚合外的对象不能持有聚合内的对象的引用，聚合内的对象可以持有其他聚合内的对象的引用。
◎ 避免在聚合内使用依赖注入。

（9）工厂（Factory）：工厂用来封装对象的创建所必需的信息，对创建聚合特别有用。

一个对象的创建可能是它自身的主要操作，但是复杂的组装操作不应该成为被创建对象的职责，因为组装这样的职责会产生笨拙的设计，也很难让人理解。而工厂可以帮助封装复杂对象的创建过程，并且当聚合根建立时，所有聚合包含的对象也随之建立了，整个过程是又是原子化的。

（10）仓储（Repository）：仓储是对聚合的管理，它介于领域模型和数据模型之间，主要用于聚合的持久化和检索，同时对领域模型和数据模型进行隔离，以便我们关注领域模型而不需要考虑如何进行持久化。肯定有人问：“为什么不能直接使用数据访问层（比如 ORM 框架等），而需要多加一个仓储层来解耦呢？”下面主要介绍仓储与数据访问层的区别。

◎ 仓储限定了只能通过聚合根来持久化和检索领域对象，以确保所有改动和不变性都由聚合根处理，保证聚合内的一致性。

◎ 仓储通过隐藏聚合持久化和检索的底层技术实现，从而达到与具体实现的解耦能力，即领域层不需要知道通过什么方式来持久化领域对象。比如当前使用的是 Hibernate，而未来想改为 MyBatis，这样的变动对于领域层来说是完全透明的。

◎ 仓储为数据模型和领域模型定义了一个边界，解决了领域模型和数据模型的混乱情况（很多人都错误地把 ER 数据模型当作领域模型）。

虽然以上要素的责任不一样，所负责的层级也不同，但是各要素之间并非独立，其关系如图 3.30 所示。

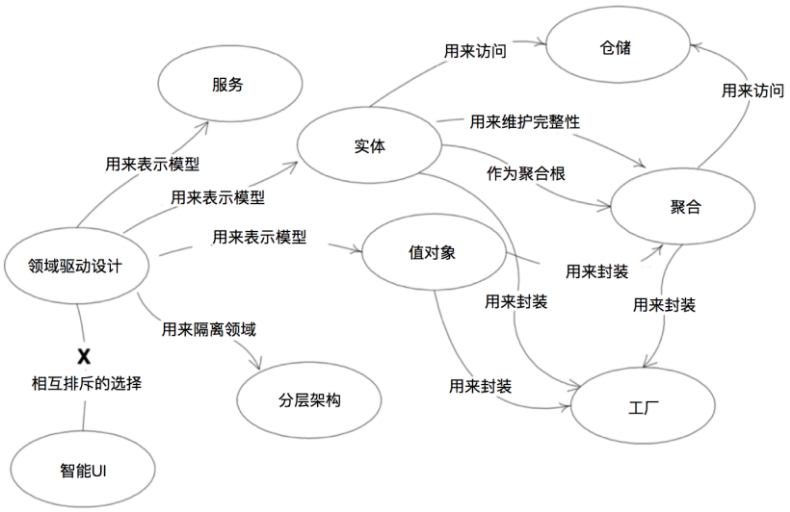


图 3.30

3.8 领域模型实践

领域驱动设计中的核心内容是领域模型，本节就来讲解领域模型实践。

3.8.1 什么是领域模型

领域模型（Domain Model）是对现实世界或者领域中对象的可视化表示。领域模型也被分为概念模型、领域对象模型和分析对象模型。领域模型也叫作问题域模型，表述的是某个领域的现实概念。

经常有人把物理数据模型（比如数据库的 ER 模型）和领域模型混为一谈，其实这是一个误区。物理数据模型在本质上归属于结果域模型（Solution Space Model），是对某个问题域的解决方案的一个具体描述。而领域模型是为了准确定义需要解决的特定问题而构造的抽象模型，其最重要的功能是形成统一的认知，也就是说所有相关人员都需要对要解决的问题有一个完整、规范和一致的认知。

我们经常在公司听到一句话：一颗心，一张图，一场仗，指我们首先要明白解决什么问题（一场仗），才能把这个问题毫无歧义地表述成统一的模型（一张图），有了这模型，我们就能全心全意地投入到解决这个问题的过程中，达到对问题的统一认知（一颗心）。所以，如果大家在工作中经常争论一个问题，则很可能是因为大家对问题的定义还没有统一的认知。而系统、科学、有逻辑地定义我们的问题，从而达到对要解决的问题的统一认知，就是领域建模的过程。

领域建模用于对领域内的概念或现实世界中的对象进行可视化表示，专注于分析问题域本身，发掘重要的业务领域概念，并建立业务领域概念之间的关系，所以它是对业务角色和业务实体之间应该如何联系和协作以完成业务功能的一种抽象。

3.8.2 领域模型的作用

领域模型是对业务概念的可视化描述，是需求分析的产物，用于指导程序设计，但领域模型与实现方式无关，在领域建模时不应该考虑如何实现，需要同项目的所有成员（客户、项目经理、开发、测试等）达成共识。

为什么要做领域建模？首先，从 20 世纪 80 年代开始，人们对于面向对象建模就已产

生许多思考和方法，其中最流行的就是面向对象分析与设计，面向对象分析强调的是在问题域发现并描述概念，解决的问题是做正确的事情；面向对象设计强调的是定义软件对象，解决的问题是正确地做事情。然后，建模的重要性在所有工程实践中都已经得到了广泛的认同，它是一种抽象和分解的方法，可以将复杂的问题拆解成一个个的抽象块，代表特定的一块密集而内聚的信息。最后，领域模型就是面向对象分析的主要产物，表达了对现实问题的描述和抽象，如图 3.31 所示。

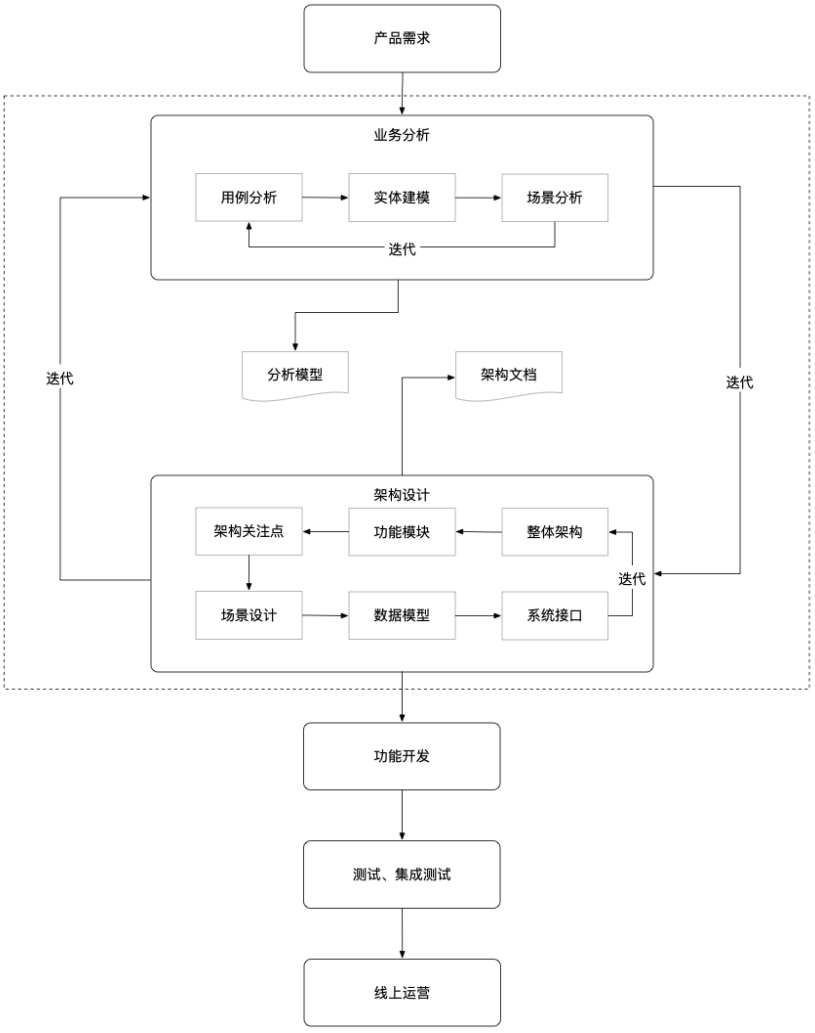


图 3.31

如果不做设计，直接实现，则很有可能在开发过程中发现思维的局限性，使开发推倒重来；并且，通过这种方式构造的代码，并没有和现实世界连接起来，当我们的软件和需求稍加修改时，代码就可能变得异常混乱和难以维护。而通过领域建模的自上而下的设计，可以保证代码实现的层次结构和模块划分是科学、稳定的。

同时，随着时间的迁移，系统的演变导致系统之间的依赖越来越复杂，因为模型设计与现实世界变得不一致，模块职责变得越来越不明确，系统之间的交互变得越来越复杂，最终整个系统都变得难以维护。

举个例子，在某电商系统的初期设计中，一个卖家账号只能开一个店铺，因此卖家和店铺的概念全部由卖家这一个模型来承载。所有与店铺相关的模型（店铺红包、店铺评价等）全部与卖家模型关联。这时，同一个模型拥有了两层业务含义，职责不明确。

领域建模可以降低软件和现实世界之间的差异，用真实的业务概念划分职责，目的是实现一个可以高效率、低成本维护的可持续发展的软件系统。

从领域模型推导到系统实现是一套引导思考的方式，也是一套科学的开发流程，其核心目的在于提供系统设计的“指导方针”。领域模型必须基于用户需求和业务发展，既可以用来同用户沟通验证需求，又可以避免模型因实现的考量而被带偏（实现成本、遗留系统）。

3.8.3 如何进行领域建模

领域建模的方法也有很多种。下面列出的是一些常见的方法。需要注意，领域建模是需要依赖大量经验和思考的，各种方法只起引导的作用。

1. 用例分析法

用例分析法是进行领域建模最简单可行的方式，其步骤如下。

（1）获取用例描述。既然我们的领域模型指的是问题域模型，那么建模也一定要从问题域入手。那么问题域的知识是如何表现出来的呢？一种最常见的方式就是通过用例，也可以通过场景来分析，总之就是一段格式化的需求文字描述。

（2）寻找概念类。寻找概念类就是对获取的用例描述进行语言分析，识别名词和名词短语，将其作为候选的概念类。当然，需求描述中的名词不可能完全等价于概念类，自然语言中的同义词、多义词都需要在此处进行区分。还有很多名词可能只是概念类的属性，

不过没关系，在这一步都可以提取出来，在第 4 步再区分出概念类和属性。

（3）添加关联。关联意味着在两个模型之间存在语义联系，在用例中的表现通常为两个名词被动词连接起来；所有动词关联的概念类并非都需要作为关联存在，更重要的是我们需要判断，两个概念类的关系是否需要被记住；试想你是一个业务员，如果某两个概念类的实例之间的关系没有任何人知道，是否会阻碍业务的开展？如果答案是肯定的，那么说明这两个概念类存在关联；如果答案是否定的，那么建议不要加上关联（视情况而定，也要考虑逻辑上二者的关系是否“被记住”）。应该尽量避免加入大量的关联，关联不代表数据流，也不代表系统调用关系。

（4）添加属性。对于上文抽取的名词列表，我们需要区分概念类和属性（当然，名词列表也会有无用的词语）。如何判断一个名词是否是属性？能完全通过基本数据类型（数字、文本、日期）表达的大多是属性。如果一个名词只关联一个概念类，并且它自身没有属性，那么它就是另一个概念类的属性。注意，这里表达的依然是业务概念，外键 ID 不是属性。

（5）模型精化。这是可选的步骤，有时我们希望在领域模型中表达更多的信息，这时会利用一些新的手段来表达领域模型，包括泛化、组合、子域划分等领域模型都可以使用 UML 的泛化和组合来表达模型间的关系，它们表达的是概念类的“IS-A”和“HAS-A”的关系，并不是实现的软件类的关系。然而，在一些方法论中并不建议使用这种方式来表达领域模型，因为这种精化的领域模型不利于和需求方沟通。子域划分是常见的拆解领域的方式，通常来说，我们会将更内聚的一组模型划分为一个子域，形成更高一层的抽象，利于系统地表达和分工。

2. 领域驱动设计建模的步骤

领域驱动设计本身是一套完整、详尽的方法论，从如何进行需求沟通（构建领域知识），到高层设计（战略建模）、详细设计（战术建模），细致到代码的实现风格都给出了示例。需要再次强调的是，领域驱动设计的模型在本质上是 Solution Space 的模型，然而领域驱动设计强调模型与实现绑定，因此这里指的“模型”可以说是领域模型，也可以是系统模型。下面就是领域驱动设计建模的一般步骤。

（1）构建领域知识。软件的最终目的是增加一个特定的领域。为了达到这个目的，软件需要与自己要服务的领域“和谐相处”，软件需要精确反映领域的概念和知识，以更好地适应变化。因此，软件开发者第一步且最重要的一步就是理解领域知识。领域驱动设计

鼓励开发者和领域专家一起工作，通过交谈和提问，让开发者学习到领域知识，挖掘出领域的关键概念。

（2）创建通用语言。通用语言是领域专家和开发团队之间定义的标准术语，目的是把领域知识更完善地传达到软件中。团队在进行所有方式的沟通时（文字、演讲、图形）都需要采用这种一致的语言。通用语言需要映射到模型和代码里，对通用语言的更改就是对模型的更改，也是对代码的更改。

（3）创建实体。基于通用的语言和领域知识，我们首先需要分辨出实体。实体具有标识唯一性，两个实体若状态一样但标识不一样，就是两个不同的实体。实体通过属性来描述。

（4）创建值对象。值对象不具有唯一性，如果两个对象的状态一致，它们就是同一个值对象。我们也可以将值对象理解为一组聚合的属性，例如地址信息、类目信息等。

（5）创建聚合根。聚合根是一个实体，将一组模型聚合在一起，与外部模型划分开来。这一组模型全部关联着聚合根，只有聚合根负责与外部的访问。聚合根还有助于保持领域模型关联关系的简化和生命周期的维护。

3. 四色建模法

四色建模法源于 *Java Modeling In Color With UML*，是一种基于模型的分析与设计方法，通过把所有模型都分为 4 种类型，使模型清晰、可追溯，如图 3.32 所示。

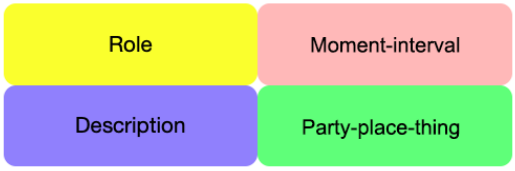


图 3.32

四色模型如下。

◎ Moment-interval（时标性原型）：时标性原型是建模的起点，代表着我们需要记录的某一时刻发生的事件，例如订单、行程和会议。

◎ Party-place-thing（人-事-物原型）：是一种有形的可唯一识别的实体，可以是人、机构、地点、物品等。

◎ Role（角色原型）：角色是人、事、物的一种参与方式。例如，在一份雇用关系中，某个人扮演者雇员的角色，那么这个人就是人-事-物原型，雇员就是“Role”。

◎ Description（描述原型）：表示资料类型的资源，是一种类似目录条目的描述，用来对对象进行分类或标记，可以被其他原型反复使用。例如，一个商品的品牌、描述和属性。

建模次序如下。

（1）在满足运营、管理需求的同时支持事件追溯。

（2）根据需求追溯，找出相应的时标性原型。

（3）寻找时标对象周围的人-事-物。

（4）从中抽象角色。

（5）把一些信息用描述对象补足。

4. 从领域模型到系统模型

提到模型，大多数人的第一反应就是一个类或者对象，这里指的系统模型就是这种概念。

为了保证程序的实现遵循领域模型的思想，并让所有人都对领域和职责的认知没有偏差，这里强烈建议每个领域模型都要有一个系统模型与之对应，最好能完全一一对应（领域驱动设计就是这么做的），它们的命名和属性也尽可能保持一致，使用相同的术语。具体到系统模型的设计，就是面向对象设计的范畴了，可以使用各种各样的设计模式、GRASP、SOLID 去设计和规划，这里不再赘述。需要遵循的宗旨是，领域模型的模型职责、子域边界划分应该作为此处设计的指导原则，每个模块都不可以突破这些职责的约束。

2

第 2 篇

架构修炼

第 4 章

架构思维

软件架构是一个系统的草图，描述了组成架构的组件及各个组件之间的关系，组件和环境之间的关系，以及设计组件的原则。组件可以是子系统、模块、类、方法等。

架构设计是架构决策的过程，涉及系统分解、接口定义、通信协议定义、交互关系和集成方式确定。架构决策指在架构设计中统筹全局并做出决定、权衡和取舍，比如将系统拆分为几个子系统，子系统的职责是什么，子系统之间如何交互，如何调用和采用什么集成机制，使用什么开发语言和技术框架。

软件架构根据定义可分为组成派和决策派。

（1）组成派。软件架构是计算机组件与组件之间的交互，有如下两个显著的特点。

◎ 关注软件本身，以软件本身为描述的对象。
◎ 通过分析软件的组成来说明软件不是一个简单的整体，而是由不同的组件通过对接口的约定连接成的一个整体。

（2）决策派。指软件架构包含一系列决策，如下所述。

◎ 软件系统的组织。
◎ 选择组成系统的组件和组件之间的接口，以及组件之间协作时的所有行为。
◎ 如何将这些组件集成在一起，形成更大的子系统。

4.1 分解

在软件架构中，分解是一种非常重要的手段，是“分而治之”思想的体现。“分而治之”是一种处理复杂问题的通用方法，能保证分解后的各个部分高内聚、松耦合，最终集

成为一个整体。多层架构、OSI 七层模型等都体现了“分而治之”的思想。在架构设计过程中，会通过将关注点分离对架构进行多层次分解，将系统层层分解为多个架构元素，进而识别架构元素。可以说，架构中的很多元素都是在架构分解的过程中产生的。

4.1.1　分解的作用

万事开头难，架构分解是架构设计过程中非常关键的一步。除了识别架构元素，对于大型软件系统而言，分解还是解决非功能需求的重要手段。为了解决可伸缩性、可用性和可管理性等问题，可在架构的多个层面进行分解。

（1）在应用层面，按照功能或 SOA 服务进行分解，将系统垂直拆分为多个应用池（应用池中的服务是无状态的），在每个应用池中都有多个应用（水平拆分），可以独立、灵活地进行伸缩。

（2）在数据层面，对数据进行垂直拆分（分库）和水平拆分（即数据分片，DB Sharding），将分布式事务拆分成多个本地事务独自提交，避免分布式事务。

4.1.2　分解的原则

德国哲学家、数学家莱布尼茨一针见血地指出：“不讲究分解技巧，分而治之的作用就不大。无经验者对问题分解不当，反而增加了解决问题的困难。”为了正确地进行分解，需要遵循一些原则，如下所述。

（1）高内聚、低耦合。莱布尼茨指出：“分解的主要难点在于怎么分。分解策略之一是按容易求解的方式来分，之二是在弱耦合处下手，切断联系。”高内聚、低耦合也是软件设计的基本原则，其实，我们可以将软件设计中的很多设计原则都看作它的派生或具体化，例如单一职责原则、依赖倒置原则和模块化封装原则，这些原则在架构分解中也是适用的。

（2）层次性。分解通常是先业务后技术，循序渐进，先逻辑后物理，从上到下逐级进行分解和展开，一般是系统→子系统→模块→组件→类。

（3）正交原则。和物理学中的正交分解类似，架构分解出的架构元素应是相互独立的，在职责上没有重叠。

（4）抽象原则。架构元素识别在较大程度上是架构师抽象思维的结果，架构师应该具

备在抽象概念层面进行架构构思和架构分解的能力。

（5）稳定性原则。将稳定部分和易变部分分解为不同的架构元素，稳定部分不应依赖易变部分。根据稳定性原则，将通用部分和专用部分分解为不同的元素；将动态部分和静态部分分解为不同的元素；将机制和策略分离为不同的元素；将应用和服务分离。

（6）复用性原则。就是对之前成果的复用，复用类似的架构设计、分析设计、开源框架、设计模式、数据结构、领域模型、架构思想等，已有的这些成果已经在不同纬度上分解并识别出了很多架构元素、或指出了分解的方式和方法，可以帮助我们快速分析和分解现有需求和架构。

4.1.3 分解的时机

架构分解的时机通常就是架构改造和演变的时机。架构在腐化时，已经难以满足关键场景的关键需求，例如对用户请求的响应速度越来越慢，已经接近临界值，并且根据经验，响应速度还有可能继续降低，越来越难以维护，这时可以考虑进行架构演变，对架构进行改造和分解。当然，如果能提前预见系统的问题，在经过慎重评估后，提前进行架构演变，则也是可以的。

注意，不要过早分解、过度分解，这样做除了增加成本，还可能带来风险。例如，很多系统在建设初期考虑到规模较小和快速上线，通常都会打造一个整体的系统，不会进行大的架构分解，之后随着需求和规模的逐渐增大，会逐步进行架构改造和架构分解。

4.2 集成

集成是配合分解完成的动作。分解完成的各个组件或子系统，通过合适的接口设计，最终还能够集成为一个完整的整体。分解的目的仅仅是加速开发和降低问题的复杂度，如果分解后的内容无法集成在一起，分解就没有任何意义。我们可以将“分解+集成”理解为架构的核心思考方式和方法。

4.2.1 常见的集成方式

在 SOA 体系下，服务之间通过 ESB 通信，许多业务逻辑存在于中间层（消息的路由、转换和组织）。

微服务架构倾向于减少中心消息总线（类似于 ESB）的依赖，将业务逻辑分布在每个具体的服务终端。大部分微服务基于 HTTP、JSON 这样的标准协议，集成不同的标准和格式变得不再重要；并且，在微服务架构中还可以采用轻量级的消息总线或者网关，有路由功能，没有复杂的业务逻辑。

下面介绍几种常见的集成方式。

1. 点对点方式

在点对点方式中，服务之间直接调用。每个微服务都开放 Rest API，并且调用其他微服务的接口，如图 4.1 所示。

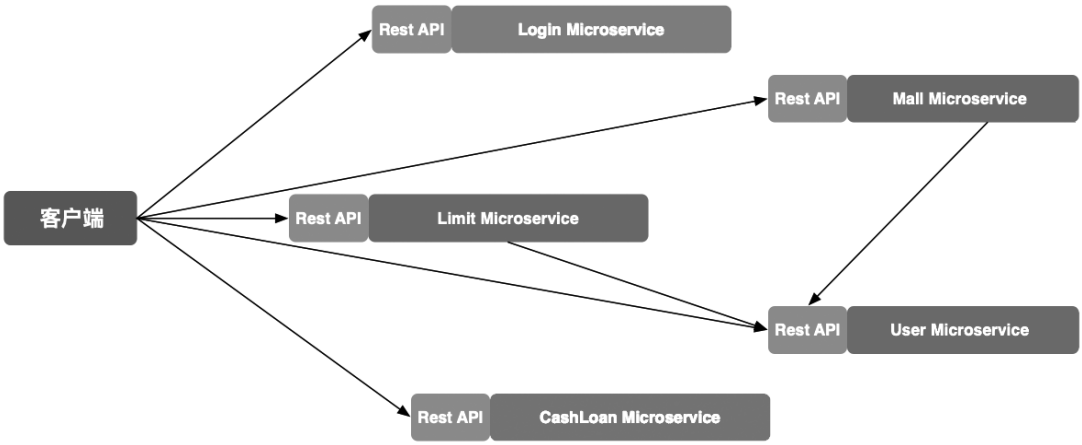


图 4.1

将这种方式应用于简单的微服务架构是完全可行的，但是当微服务架构越来越复杂时，这种方式就不适用了。这有些类似于 SOA 的 ESB，所以我们尽量不采用点对点的集成方式。

2. API 网关方式

API 网关方式的核心要点是，所有客户端和消费端都通过统一的 API 网关接入微服务，在 API 网关层处理所有的非业务功能。通常，API 网关也提供 Rest、HTTP 的访问 API，服务端通过 API-GW 注册和管理服务。

举个例子，在某个项目中，所有业务接口都通过 API 网关暴露，是所有客户端接口的

唯一入口，微服务之间的通信也通过 API 网关进行，如图 4.2 所示。

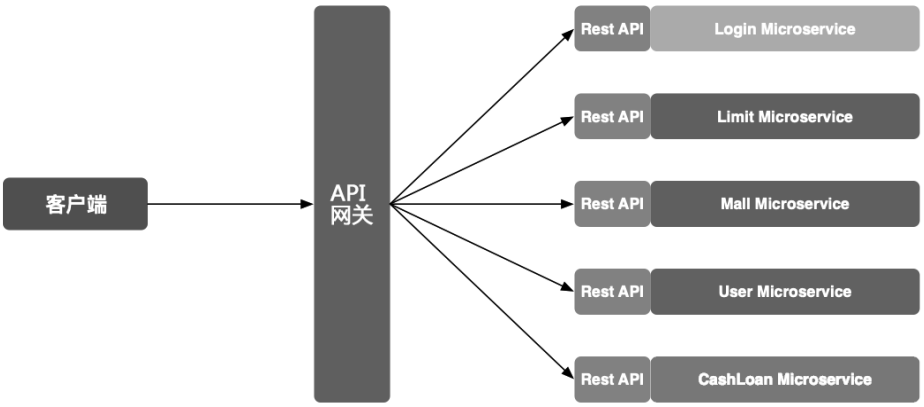


图 4.2

可以看出，API 网关方式有如下优势。

◎ 为微服务的接口提供一个统一的抽象网关层，比如微服务提供的接口有很多类型，在网关层提供了统一规范的接口。
◎ 轻量的消息路由、格式转换功能。
◎ 统一控制安全、监控、限流等非业务功能。
◎ 每个微服务都变得更加轻量，非业务功能都在网关层统一处理，微服务只需要关注业务逻辑。

3. 消息代理方式

微服务也可以通过异步方式来集成，通过队列和订阅主题实现消息的发布和订阅，一个微服务可以是消息的发布者，将消息通过异步方式发送到队列或者订阅主题下。作为消费者的微服务可以从队列或者主题中获取消息，通过消息中间件将服务之间的直接调用解耦，如图 4.3 所示。

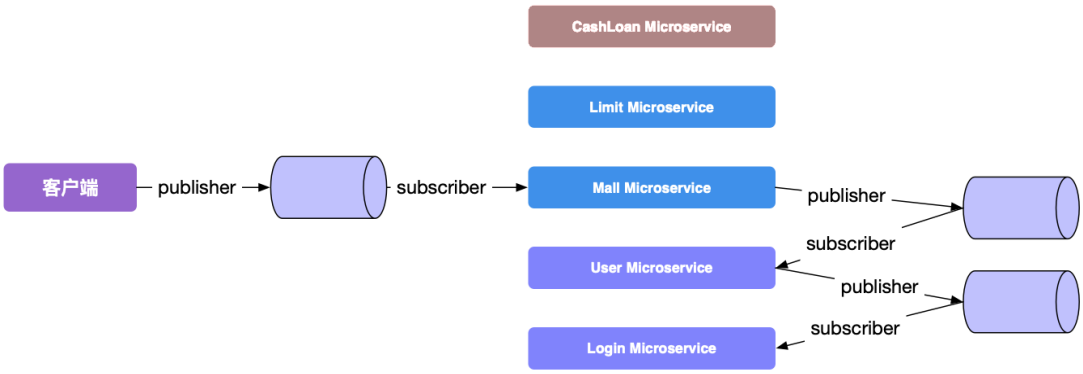


图 4.3

如图 4.3 所示，异步的生产者/消费者模式通常通过 AMQP、MQTT 等异步消息进行规范。

4.2.2　集成的难点

微服务架构变得越来越流行了，还成为模块化的一种方法，将一整个应用拆分成一个个服务，让团队在开发大型复杂系统时，能更快地交付高质量的项目，并且可以轻松接受新技术，因为团队成员可以使用最新且流行的技术栈实现各自的服务，微服务架构也通过让每个服务都被部署在最佳状态的硬件上来改善应用的扩展性。

但是，在微服务集成过程中，我们需要迈过三道难关：数据服务设计；数据的一致性；分布式查询。不同于传统的单体架构、基于 ESB 的 SOA 系统的架构，在微服务化过程中产生了许多新问题，许多应用大到一个人根本无法完成，而且复杂到只靠一个人都无法理解。

在这种情况下，应用就必须被拆分成一个个模块。在单体应用中，模块被定义为一个 Java 包，这种做法在实践中效果并不很理想，时间长了，单体应用会变得越来越庞大。微服务架构将服务作为一个模块单元，每个服务都有一个不可渗透且很难逾越的边界，即每个微服务都要提供一种单独且独立的能力，这样的话，应用程序的模块化就更容易随时保存。

接下来看看怎么迈过以上所述的三道难关。

1. 数据服务设计

通用架构是不存在的，所以我们需要了解微服务架构下的几个关键考量点，然后针对自己的实际应用选择更重要的考量点。下面讨论从哪几个角度着手设计一个符合微服务架构原则的数据架构。

◎ 是否可以用一个数据库或者多个数据库来支持多个微服务？
◎ 如果有多个数据库，则是否需要为每个微服务都挑选一个最合适的数据库，还是选择同一种类型的数据库？
◎ 如何在微服务架构下扩展数据库？
◎ 当一个依赖的服务需要修改数据库 Schema 的时候，是否会影响到现有系统？
◎ 当微服务应用不断演变的时候，数据库是否可以快速响应应用需求的变化？

在数据服务设计遇到以上问题时，我们可以通过下面的手段来解决。

1）采用一库一服还是一库多服

无论是单体应用还是微服务应用，有一点是肯定的：在应用的各个模块之间都需要进行较为频繁的通信，通过协同合作来实现应用的整体价值。在单体应用中，这种通信是通过方法调用来完成的，在微服务中则是通过 API 调用来完成的。这些模块或者服务间的调用，在大多数时候是为了共享数据。共享数据的最简单方式就是采用一种共享数据库的模式，也就是单体应用常用的方式：应用可以有多个系统模块，但一般只有一个数据库。

这种架构模式通常被认为是微服务架构下的反范式，它的问题如下。

◎ 单点故障，一个数据库倒下，整批服务全部停止。
◎ 数据在同一个地方，随着需求量的增加，数据间的依赖越来越复杂。
◎ 无法针对某个服务进行精准优化或扩展。

所以一般推荐的做法，是为每个微服务都准备一个单独的数据库，即一库一服（DataBase Per Service）模式，这种模式更加适合微服务架构：它满足每个服务都是独立开发、独立部署、独立扩展的特性。当需要对一个服务进行升级或者数据架构改动的时候，无须影响到其他服务；当需要对某个服务进行扩展的时候，也可以手术式地对某个服务进行局部扩容。另外，如果某些服务对数据库有特殊的需求，则在这种模式也为下文所讲的混合持久化（Polyglot Persistence）提供了可能性。

2）混合持久化

混合持久化在大型互联网公司中是比较流行的模式，它秉承的原则就是为特别的任务提供最好的工具。比如，如果某个系统提供一个高并发、低延迟的共享用户会话方案（Shared Session Storage），则 Redis 可能是一个非常理想的选择；如果实现一个产品目录，涉及大量不确定结构的商品数据及属性的建模管理，则系统可能会采用模式灵活、有动态 Schema 的 MongoDB 作为数据库解决方案；如果系统希望支持非常强大的全文搜索能力，则可以采用 Elasticsearch。

混合持久化的优势很明显，可以让每个单独的服务都使用最佳的工具和技术，但它的弊端也是不容忽视的。

◎ 部署、监控、备份、升级等都是运维工作中不可或缺的内容。
◎ 引入多个不同的数据库，也意味着系统管理与维护的复杂度、成本提高了很多，在这种情况下，比较有资源的公司或者团队才可以采用混合持久化模式。

这也解释了混合持久化模式在大型互联网公司中得到较多采用与推广的原因。对于其他小规模用户，或者缺乏掌握各种新型技术人才的公司来说，另一种更为可行的模式可能是多模数据库（Multi-model）。

3）多模数据库

多模数据库的特征是：

◎ 依然是一库一服务（为一个服务部署一个单独的数据库）；
◎ 采用可以支持多种应用场景的数据库；
◎ 虽然是多实例模式，但属于同一种类型的数据库，人员组成和维护管理都比较简单。

如果正在开发的应用是一款企业级产品，会将其交付到客户环境下安装、部署，则运维管理的简单性将在技术选型中占据非常大的比重，在这种情况下，多模数据库更加适用。

2. 数据的一致性

单体应用可以通过 ACID 事务来实现数据的一致性。比如个人消费金融系统的用户都有信用额度，应用程序在创建订单前会先看用户的信用额度如何，必须保证在有多个并发尝试创建订单时不超过客户的信用额度。如果订单表和用户表都在同一个库中，就可以通过 ACID 事务来完成。但是在微服务系统中无法通过这样的方式来实现数据的一致性，因为订单表和用户表都在各自的应用服务中，只能通过各自的服务 API 访问，它们甚至可能存在于不同的网络中。

比较常见的做法是使用分布式事务来完成，比如 2PC（二阶段提交）等，但这对于现在的应用来说可能并不可行。由于系统设计要遵循 CAP 定理，所以系统必须在可用性和一致性之间进行选择，可用性通常是较好的选择，而且现在的很多技术，例如大多数 NoSQL 数据库甚至不支持 ACID 事务，更不用说 2PC。所以在实现数据的一致性时要采用其他方式。我们使用事件驱动架构中的一种事件源（Event Sourcing）技术来应对数据一致性的问题。

事件源是通过使用不同的事件中心来获得不需要 2PC 的原子性，保证业务对象的一致性。这种应用存储业务对象的一系列状态改变事件，而不是存储对象现在的状态。应用可以通过重放事件来重建对象的现在状态，只要业务对象发生了变化，新事件就会被添加到时间表中。因为保存事件是单一操作，因此肯定是原子性的。

为了理解事件源的实现原理，我们考虑将事件对象作为一个例子。在传统方式中，每个订单都被映射为订单表中的一条记录，例如订单商品表。对于事件源方式，订单服务基于事件状态的改变来存储一个订单，比如创建订单、待支付、已支付、待发货、已发货、取消订单，每个事件都包括足够的数据来重建订单状态。

事件状态被长期保存在事件服务的数据库中，通过 API 添加和获取实体事件的功能列表。事件存储和之前描述的消息代理模式类似，通过 API 来订阅事件。事件存储将事件传送给对事件感兴趣的所有订阅者，是事件驱动微服务架构的基础。

事件源的优点如下。

◎ 解决了事件驱动架构的关键问题，使得只要有状态变化就可以可靠地发布事件，也就解决了微服务架构中数据的一致性问题。
◎ 因为是持久化事件而不是对象，所以避免了面向对象编程与关系型数据库间的不一致性。
◎ 事件源方法提供了 100%可靠的业务实体变化监控日志，使得获取任何时点的实体状态都成为可能。
◎ 事件源方法可以使业务逻辑由事件交换的松耦合业务实体构成。

以上的优点保证了将单体应用移植到微服务架构变得相对容易。

事件源缺点如下。

◎ 因为采用了不同的或者人们不太熟悉的编程模式，所以重新学习其编程模式不太容易。

◎ 事件存储只支持主键查询业务实体，必须使用命令查询的责任分离（Command Query Responsibility Segregation）来完成查询业务，应用需要通过最终一致性来保证数据的一致性。

3. 分布式查询

在单体系统中一般使用 Join 语句实现多表查询。比如，可以通过 SQL 语句轻松查询出客户最近所订的大额订单。但是，我们无法在微服务架构中实现这样的查询。就像前面提到的那样，Orders 和 Customers 表分属不同的服务，只能通过服务 API 来访问，而且它们可能使用了不同的数据库，使用事件源更新和查询问题可能更麻烦。

使用命令查询的责任分离模式是实现查询的最好方法。命令查询的责任分离模式将应用程序分为如下两部分。

（1）命令侧（command-side）。处理命令（例如 HTTP POST、PUT 和 DELETE）可以创建、更新和删除聚合，前提是这些聚合是使用事件源实现的。

（2）查询侧（query-side）。通过查询聚合的一个或多个物化视图来处理查询（例如 HTTP GET），通过订阅由命令侧发布的事件保持视图与聚合同步。查询侧的视图可以使用任意类型的能满足需求的数据库来实现。根据需求，应用程序的查询侧可能使用一个或多个数据库。

在很多场合下，命令查询的责任分离模式都是一个以事件为基础的综合体，比如使用 RDBMS 作为记录系统，使用 Elasticsearch 查询文本，查询侧可以使用其他类型的数据库，支持多种类型的数据库，不仅仅是文本搜索引擎，而且通过订阅事件准实时地更新查询侧的视图。

命令查询的责任分离模式的优点如下。

◎ 在微服务架构中实现了查询，特别是使用事件源的架构，使应用程序有效地支持一组不同的查询。
◎ 将命令侧和查询侧分离，达到了解耦的目的。

命令查询的责任分离模式的缺点如下。

◎ 需要额外的工作来开发和维护这套系统，需要开发和部署更新视图、查询视图的查询端服务，还需要部署视图数据库。
◎ 在命令侧和查询侧的视图之间有“滞后”。查询侧与命令侧存在一定的时延，在数

据更新、聚合后，查询到的可能依然是聚合之前的数据，这时就需要通过其他手段强制更新，使查询侧和命令侧的数据保持一致。

4.3 动静分离

动静分离是最重要的架构思维之一，是将静态资源与动态资源分离，通过不同的系统进行访问的架构设计方法，在设计架构时一定要注意二者的结合，因为既涉及流程、用例等动态分析内容，又涉及数据、类等静态建模内容，需要高度协作来完成。

动静分离的适用场景如下。

◎ 页面对应的 URL 通常固定。
◎ 在页面中不包含时间因素、地域因素，以及浏览者相关的因素。
◎ 页面访问量大，服务器负载高、I/O 等问题导致用户访问卡顿。
◎ 页面数量大，导致服务器存储空间不够。
◎ 某个文件在特定的时间段内需要高速下载，且并发下载量高。
◎ 不能包含 Cookie 等私有数据。

动静分离的使用场景如图 4.4 所示。

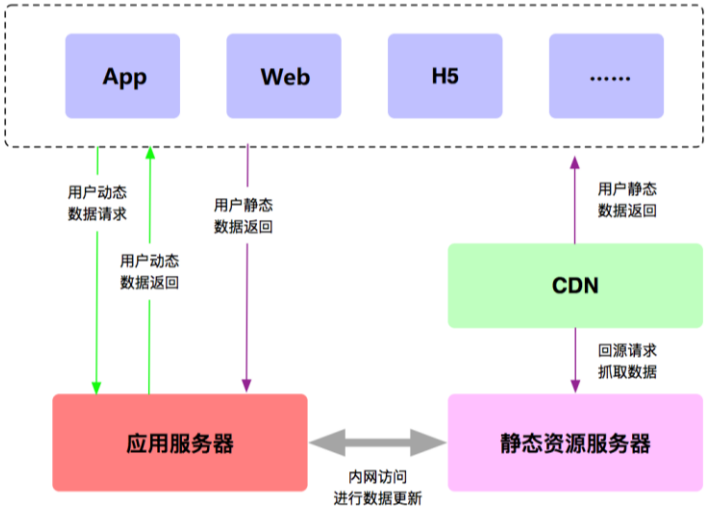


图 4.4

可以看出，动静分离的好处如下。

◎ 减少 Web 服务器负载，静态文件全部通过 CDN 进行访问。
◎ 采用价格低廉的云存储服务器，存储费用最低。
◎ 海量存储空间，无须考虑存储架构的升级。
◎ 主要使用 CDN 流量，流量费用低。

4.4　复用

复用是另外一种重要的架构思维，也可以将其理解为 SOA 参考架构的核心思考模式，包括最近广受关注的业务能力组件化、组件能力服务化、平台+应用、共享中心建设、共性能力下沉等都是复用的表现。即使设计一个小系统，也需要将各个模块需要用到的共性功能抽取为可复用的共性组件。

好的系统设计具备可扩展性（Extensibility）、灵活性（Flexibility）和可插入性（Pluggability）。一个复用性较好的系统，就是一个易维护的系统。但实际上，可维护性和可复用性是两个维度的内容。

（1）系统维护：系统维护就是系统的再生。一个可维护性较好的系统，必须允许新的元素以比较容易和平稳的方式加入已有的系统中，让这个系统不断焕发活力，其系统维护工作应该以容易、准确、安全和经济的形式进行。造成系统可维护性差的原因如下。

◎ 过于僵硬：指在系统中加入一个新的功能，不管大小都很难，这个新功能会波及很多其他模块，最后造成跨越几个模块的改动。
◎ 过于脆弱：与系统的过于僵硬同时存在的，是系统在修改已有代码时过于脆弱。修改一个地方，会导致系统到处发生故障。
◎ 复用率低：指一个系统的组成部分可以在同一个项目的不同地方甚至不同的项目中重复利用。复用率低，指已有的代码、函数、模块的功能在新的模块或系统中使用时发现依赖很多其他因素，导致很难被再次利用。
◎ 耦合度过高：实现一个需求有两种方式，第 1 种方式是不改变原有设计意图和原有架构设计，这对于未来有利；第 2 种方式是改变原有设计意图和原有架构设计，这只能解决短期的需求，只是一个权宜之计，会牺牲中长期利益。如果一个系统采用第 2 种方式比采用第 1 种方式更容易、更简单，这个系统的耦合度就很高。

（2）系统复用的重要性如下。

◎ 有较高的生产效率。
◎ 有较高的系统质量。
◎ 恰当运用复用可以改善系统的可维护性。

4.4.1 系统复用的分类

我们可以将系统复用分为常规复用和系统层复用。

常规复用可分为如下三种。

（1）代码复用。利用编辑器（IDE）可以很方便地减少抄写代码的人力成本。将一个写好的工具函数自发地复制到另一个功能或项目中，可能是入门级程序员习惯性的自发的复用软件形式，这种方式有着明显的缺点，会造成很多代码的重复、冗余。

（2）算法复用。各种算法如排序算法都已经得到了大量的研究，几乎不需要我们重写自己的算法，各种语言通常也实现了这些常用算法，因此直接复用即可。

（3）数据结构的复用。与算法一样，类似数组、队列、栈、列表等都得到了人们的深入研究，直接复用即可。

系统层复用可分为如下两种。

（1）设计复用。设计复用是比代码、算法、数据结构复用更高级的复用，受实现环境的影响较少，使可复用的组件被复用的机会更多，所需的修改更少。这种复用有如下三种途径。

◎ 从现有系统的设计结果中提取一些可复用的设计组件，并将这些组件应用于新系统的设计中。
◎ 将一个现有系统的全部设计文档都在新的系统上重新实现，也就是将一个设计运用于多个具体的实现中。
◎ 与任何应用都无关，独立设计和开发可复用的设计组件。

（2）分析复用。分析复用是比设计复用更高级别的复用，可复用的分析组件是针对业务领域中某些设计或问题抽象出的组件，受设计技术及实现条件的影响很少，所以可复用的机会更大，复用的途径也有三种。

◎ 从已有系统的分析结果中提取可复用的组件用于新系统的架构设计。

◎ 用一份完整的分析文档作为输入，产生针对不同系统和其他实现条件的多项设计。
◎ 与任何应用都无关，独立设计和开发可复用的分析组件。

4.4.2　可复用性和可维护性的关系

可复用性和可维护性的关系如下。

◎ 适当地提高系统的可复用性，可以同时提高系统的可维护性，就是在保持甚至提高系统的可维护性的同时，实现系统的复用。
◎ 适当地提高系统的可复用性，可以同时提高系统的可扩展性。系统的可扩展性由“开-闭”原则、里氏代换原则、依赖倒转原则和组合/聚合复用原则保证。
◎ 适当地提高系统的可复用性，可以同时提高系统的灵活性。系统的灵活性由“开-闭”原则、迪米特法则、接口隔离原则保证。
◎ 适当地提高系统的可复用性，可以同时提高系统的可插入性。系统的可插入性由“开-闭”原则、里氏代换原则、组合/聚合复用原则和依赖倒转原则保证。

4.4.3　可维护性地复用的设计原则

在系统设计中，在支持可维护性的同时，提高系统的可复用性是一个至关重要的问题，如何同时提高一个系统的可维护性和可复用性是面向对象设计需要解决的核心问题之一。在面向对象设计中，可维护性地复用是以设计原则为基础的。每个原则都蕴含一些面向对象设计的思想，可以从不同的角度提升一个系统架构的设计水平。

1. “开-闭”原则：OCP

任何系统都需要面临一个很重要的问题，即它们的需求会随时间的推移而发生变化。当系统需要面对新的需求时，我们应该尽量保证系统的设计框架是稳定的。如果一个系统设计符合“开-闭”原则，那么可以非常方便地对系统进行扩展，而且在扩展时无须修改现有代码，使得系统在具备适应性和灵活性的同时，具备较好的稳定性和延续性。随着系统的规模越来越大，系统的寿命越来越长，系统的维护成本也越来越高，设计满足“开-闭”原则的系统也变得越来越重要。

为了满足“开-闭”原则，需要对系统进行抽象化设计，抽象化是“开-闭”原则的关键，可以为系统定义一个相对稳定的抽象层，将不同的实现行为移至具体的实现层中完成。在很多面向对象编程的语言中都提供了接口、抽象类等机制，可以通过它们定义系统的抽

象层，再通过具体的类进行扩展。如果需要修改系统的行为，则无须对抽象层进行任何改动，只需增加新的具体类来实现新的业务功能即可，实现在不修改已有代码的基础上扩展系统的功能，达到“开-闭”原则的要求。

2. 里氏代换原则：LSP

在系统中将一个基类对象替换成它的子类对象时，程序不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象，那么它不一定能够使用基类对象。里氏代换原则是实现“开-闭”原则的重要方式之一，由于在使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型定义对象，在运行时再确定其子类类型，用子类对象来替换父类对象。

在使用里氏代换原则时需要注意如下几个问题。

◎ 子类的所有方法都必须在父类中声明，或子类必须实现在父类中声明的所有方法。根据里氏代换原则，为了保证系统的扩展性，在程序中通常使用父类进行定义，如果一个方法只存在于子类中，在父类中不提供相应的声明，则无法在以父类定义的对象中使用该方法。

◎ 尽量将父类设计为抽象类或者接口，让子类继承父类或实现父类接口，并实现在父类中声明的方法，在运行时用子类实例替换父类实例。我们可以很方便地扩展系统的功能，无须修改原有子类的代码，在增加新的功能时可以通过增加一个新的子类来实现。里氏代换原则是“开-闭”原则的具体实现手段之一。

3. 依赖倒转原则：DIP

依赖倒转原则要求我们在程序中传递参数时或在关联关系中，尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不用具体类做这些事情。为了确保该原则的应用，一个具体类应当只实现接口或在抽象类中声明过的方法，而不用给出多余的方法，否则将无法调用在子类中增加的新方法。

在引入抽象层后，系统将具有很好的灵活性，在程序中尽量使用抽象层进行编程，而将具体类写在配置文件中，这样一来，如果系统行为发生了变化，则只需对抽象层进行扩展，并修改配置文件，而无须修改原有系统的源代码，在不修改的情况下扩展系统的功能，满足“开-闭”原则的要求。

在实现依赖倒转原则时，我们需要针对抽象层编程，将具体类的对象通过依赖注入的方式注入其他对象中。依赖注入指当一个对象要与其他对象发生依赖关系时，通过抽象来注入所依赖的对象。常用的注入方式有如下三种。

◎ 构造注入：指通过构造函数传入具体类的对象。
◎ 设值注入（Setter 注入）：指通过 Setter 方法传入具体类的对象。
◎ 接口注入：指通过在接口中声明的业务方法传入具体类的对象。这些方法在定义时使用的是抽象类型，在运行时再传入具体类型的对象，由子类对象覆盖父类对象。

4. 接口隔离原则：ISP

当一个接口太大时，我们需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。每个接口都应该承担一种相对独立的角色，不干不该干的事，该干的事都要干。这里的“接口”往往有两种不同的含义：一种是指一个类型所具有的方法特征的集合，仅仅是一种逻辑上的抽象；另外一种是指某种语言具体的“接口”定义，有严格的定义和结构，比如 Java 语言中的 interface。对于这两种不同的含义，ISP 的表达方式及含义都有所不同。

◎ 当将“接口”理解成一个类型所提供的所有方法特征的集合时，这就是一种逻辑上的概念，接口的划分将直接带来类型的划分。可以将接口理解成角色，一个接口只能代表一个角色，每个角色都有其特定的一个接口，此时，这个原则可以叫作“角色隔离原则”。
◎ 如果将“接口”理解成狭义的特定语言的接口，那么 ISP 表达的意思是，接口仅仅提供客户端需要的行为，而将客户端不需要的行为隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。需要将大接口中的方法，根据其职责的不同分别放在不同的小接口中，以确保每个接口使用起来都较为方便，并都承担某个单一角色。接口应该尽量细化，接口中的方法也应该尽量少，在每个接口中只包含一个客户端（如子模块或业务逻辑类）所需的方法即可，这种机制也被称为“定制服务”，即为不同的客户端提供范围不同的接口。

5. 组合/聚合复用原则：CARP

在一个新的对象里通过关联关系（包括组合关系和聚合关系）使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用功能的目的。简

而言之，在复用时要尽量使用组合或聚合关系（关联关系），少用继承。

在面向对象设计中，可以通过两种方法在不同的环境中复用已有的设计和实现，即通过组合或聚合关系，或者通过继承，但首先应该考虑使用组合或聚合关系，因为该方法可以使系统更加灵活，降低类与类之间的耦合度，使一个类的变化对其他类造成的影响相对较少；其次考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，滥用继承反而会增加系统构建和维护的难度及系统的复杂度，因此需要慎重使用继承复用。

通过继承进行复用的主要问题在于继承复用会破坏系统的封装性，因为继承会将基类的实现细节暴露给子类，由于基类的内部细节通常对子类来说是可见的，所以这种复用又叫作“白箱”复用，如果基类发生改变，那么子类的实现也不得不发生改变；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；而且继承只能在有限的环境中使用（例如类没有被声明为不能被继承）。

6. 迪米特法则

迪米特法则（Law of Demeter，LoD）指我们在设计系统时，应该尽量减少对象之间的交互，如果在两个对象之间不必彼此直接通信，那么这两个对象就不应当发生任何直接的相互作用，如果其中的一个对象需要调用另一个对象的某一个方法，则可以通过第三方转发这个调用。简而言之，就是通过引入一个合理的第三方降低现有对象之间的耦合度。

运用迪米特法则时，要注意如下几点。

- ◎ 在类的定义中，应该尽量定义一个松耦合的类。类之间的耦合度越低，类的复用度就越高。一个松耦合的类在被修改时，所涉及的类的范围就会变小。
- ◎ 在类的声明中，每个类都应当尽量降低其成员变量和成员函数的访问权限。
- ◎ 在类的声明中，只要有可能，则都应该将一类操作设计成抽象类或接口。
- ◎ 在对象之间尽量没有直接调用关系。

4.5 分层

分层相对重要，架构在设计中要考虑分层，其核心仍然是资源+服务+应用的三大层，这样的分层仍然是 SOA 参考架构的核心思想。平台+应用更多地只是上面分层模式的一个变形。分层的目的是通过分层理清整个应用的构建过程，并保证了各层之间的独立设计和松耦合。

分层架构的一个重要原则是每层只能与位于其下方的层发生耦合。分层架构可以简单分为两种，即严格分层架构和松散分层架构。在严格分层架构中，某层只能与位于其直接下方的层发生耦合；在松散分层架构中，则允许某层与它的任意下方层发生耦合。

分层架构的好处是显而易见的。首先，层间松散的耦合关系使得我们可以专注于本层的设计，而不必关心其他层的设计，也不必担心自己的设计会影响其他层，对提高软件质量大有裨益。其次，分层架构使得程序结构清晰，升级和维护都变得十分容易，要更改某层的具体实现代码时，只要本层的接口保持稳定，对其他层就可以不必修改。即使本层的接口发生变化，也只影响相邻的上层，修改工作量小且错误可以控制，不会带来意外的风险。

要保持程序分层架构的优点，就必须坚持各层间的松散耦合关系。在设计程序时，应该先划分出可能的层次，以及此层次提供的接口和需要的接口。在设计某层时，应该尽量保持层间的隔离，仅使用下层提供的接口。

分层架构的优点如下。

◎ 开发人员可以只关注某一层。
◎ 可以很容易地用新的实现来替换原有层的实现。
◎ 可以降低层与层之间的依赖。
◎ 有利于标准化。
◎ 有利于各层逻辑的复用。

分层架构的缺点如下。

◎ 因为增加了中间层，所以降低了系统的性能，不过可以通过缓存机制来改善。
◎ 可能会导致级联的修改。这种修改尤其体现在自上而下的方向，不过可以通过依赖倒置来改善。

4.6　模式

模式是所有思维模式里面的一个重点，架构模式是一个通用的、可重用的解决方案，架构设计中的模式匹配就是将最终的业务域问题匹配到对应的技术实现上，即根据业务需求来挑选最适合的技术，而不是用主流和最先进的技术去反推业务。

架构设计模式一般有分层模式、客户端-服务器模式、主从设备模式、管道-过滤器模式、代理模式、P2P 模式、事件总线模式、MVC 模式、黑板模式和解释器模式，接下来简单介绍这些模式。

4.6.1 分层模式

分层架构的一个重要原则是每层只能与位于其下方的层发生耦合。分层架构可以简单分为两种：严格分层架构和松散分层架构。在严格分层架构中，其中一层只能与其直接下方的层发生耦合；在松散分层架构中，则允许某层与它的任意下方层发生耦合，并且要保持程序分层架构的优点，就必须坚持层间的松耦合关系。在设计程序时，应该先划分出可能的层次，以及此层次提供的接口和需要的接口。在分层设计时，尽量保持层间的隔离，上层调用下层服务只能通过接口来实现。四层架构、五层架构和六边形架构是领域驱动设计分层架构中比较经典的三种架构，下面分别对这 3 种架构进行讲解。

1. 四层架构

四层架构的模式如图 4.5 所示。

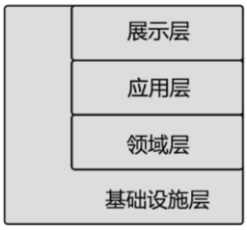


图 4.5

对如图 4.5 所示的各层解释如下。

◎ 展示层：展示用户请求返回的数据。用户可以是一个系统，不一定是使用用户界面的人。

◎ 应用层：定义系统需要完成的任务，并且指挥表达领域概念的对象来解决问题。这一层所负责的工作对业务来说意义重大，也是与其他系统的应用层进行交互的必要渠道。应用层要尽量简单，不包含业务规则或者知识，只为下一层中的领域对象协调任务、分配工作，使它们互相协作。它没有反映业务情况的状态，却可以具有另外一种状态，为用户或程序显示某个任务的进度。

◎ 领域层：负责表达业务概念、业务状态信息及业务规则。尽管保存业务状态的技术细节是由基础设施层实现的，但是反映业务情况的状态是由本层控制并且使用的。领域层是业务软件的核心，领域模型位于这一层。
◎ 基础设施层：向其他层提供通用的技术能力：为应用层传递消息，为领域层提供持久化机制，为展示层绘制屏幕组件等。基础设施层还能够通过架构框架来支持 4 个层之间的交互模式。

2. 五层架构

五层架构的模式如图 4.6 所示。

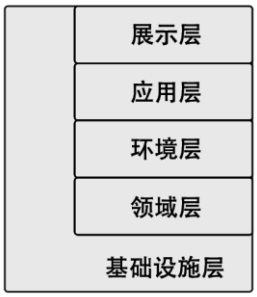


图 4.6

对如图 4.6 所示的各层解释如下。

◎ 展示层：主要用于处理用户发送的 Restful 请求和解析用户输入的配置文件等，并将信息传递给应用层的接口。
◎ 应用层：负责多进程管理及调度、多线程管理及调度、多协程调度和维护业务实例的状态模型。在调度层收到展示层的请求后，委托环境层与本次业务相关的上下文进行处理。
◎ 环境层：以上下文为单位，将领域层的领域对象转换成合适的角色，通过角色交互起来完成业务逻辑。
◎ 领域层：定义领域模型，不仅包括领域对象及其之间关系的建模，还包括对象角色的显式建模。
◎ 基础设施层，向其他层提供通用的技术能力，例如业务平台、编程框架、持久化机制、消息机制、第三方库的封装和通用算法，等等。

3. 六边形架构

六边形架构，又被称为端口和适配器风格，最早由 Alistair Cockburn 提出，并在领域驱动设计社区得到了发展和推广。这个架构之所以是六边形形式的，是为了凸显这个架构的扁平，每个边界的权重都是相等的。

六边形架构的模式如图 4.7 所示。

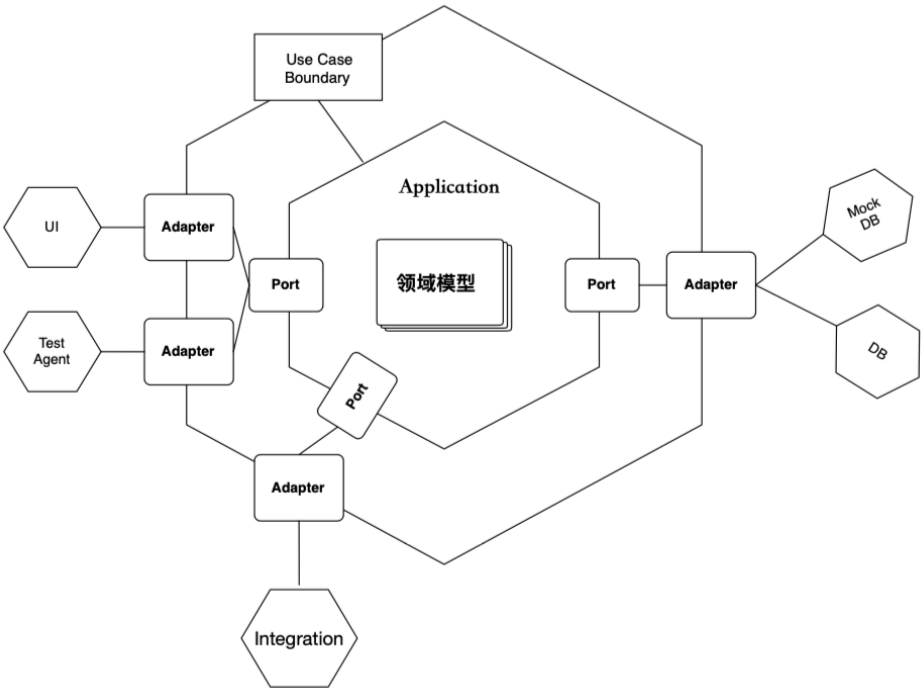


图 4.7

我们知道，经典分层架构分为三层（展示层、应用层和数据访问层），而六边形架构可以分成另外的三层，如下所述。

◎ 领域层（Domain Layer）：最内层，是纯粹的核心业务逻辑，一般不包含任何技术实现或引用。

◎ 端口层（Ports Layer）：位于领域层之外，负责接收与用例相关的所有请求，这些请求负责在领域层中协调工作。端口层在端口内部作为领域层的边界，在端口外部则扮演了外部实体的角色。

◎ 适配器层（Adapters Layer）：位于端口层之外，负责以某种格式接收输入及产生输出。比如，对于 HTTP 用户请求，适配器会将其转换为对领域层的调用，并将领域层传回的响应进行封装，通过 HTTP 传回调用的客户端。在适配器层不存在领域逻辑，它的唯一职责就是在外部世界与领域层之间进行技术性的转换。适配器能够与端口的某个协议关联并使用该端口，多个适配器可以使用同一个端口，在切换到某种新的用户界面时，可以让新界面与老界面同时使用相同的端口。

这样做的好处是使业务边界更加清晰，从而获得更好的扩展性，还使业务复杂度和技术复杂度分离，是领域驱动设计的重要基础。核心的领域层可以专注于业务逻辑而不用理会技术依赖，外部接口在被消费者调用时也不用关心业务在内部是如何实现的。

正如上文提到的，六边形架构的出现早于领域驱动设计，要实现领域驱动设计也不必非要采用六边形架构，只是这种架构正好契合了领域驱动设计所需的条件，从六边形架构出发，可以更容易理解领域驱动设计的思想。如图 4.7 所示，适配器所在的线框表示限界上下文，中间的六边形表就是应用服务，标记出核心领域模型对外暴露的功能和方法。应用层的核心就是领域模型，包含领域对象和领域服务，其中领域对象又分为实体对象和值对象，值对象中的一个特定对象就是聚合根，聚合根就是这个领域的核心。

分层模式主要用于一般的桌面应用程序和电子商务 Web 应用程序。

4.6.2　客户端-服务器模式

客户端-服务器模式简称 C/S 结构，是一种网络架构，它将客户端（Client）与服务器（Server）区分开来。每个客户端软件的实例都可以向一个服务器或应用程序服务器发送请求。客户服务器模式通过不同的途径应用于很多不同类型的应用程序，最常见的就是目前互联网上的网页。例如，当我们在维基百科上阅读文章时，我们的电脑和网页浏览器就被当作一个客户端，同时，组成维基百科的电脑、数据库和应用程序被当作服务器。当我们的网页浏览器向维基百科请求一篇指定的文章时，维基百科服务器会从维基百科的数据库中找出该文章的所有信息，并将这些信息结合成一个网页，再发送给我们的浏览器。

C/S 模式是一个逻辑概念，不是指计算机设备。在 C/S 模式中，请求方叫作客户端，响应请求方叫作服务器，如果一个服务器在响应客户端请求时不能单独完成任务，则还可能向其他服务器发出请求，这时，发出请求的服务器就成为另一个服务器的客户端。从双方建立联系的方式来看，主动启动通信的应用叫作客户端，被动等待通信的应用叫作服务器。它们的通信过程如图 4.8 所示。

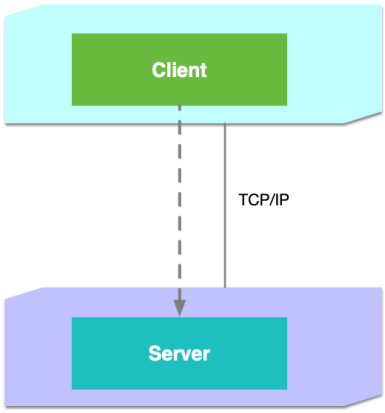


图 4.8

客户端-服务器模式主要用于电子邮件、文件共享和银行等在线应用程序中。

4.6.3 主从设备模式

主从设备（Master-Slave）模式也叫作主仆模式，它基于分而治之的思想，将一个原始任务分解为若干个语义等同的子任务，并由专门的工作者线程并行执行这些任务，原始任务的执行结果是通过整合各个子任务的处理结果形成的。主从设备模式如图 4.9 所示。

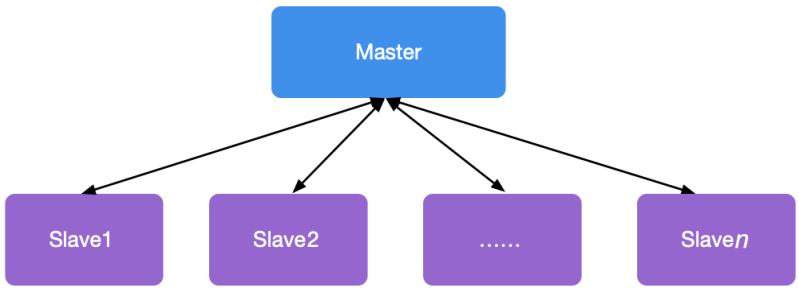


图 4.9

主从设备模式的使用场景如下。

◎ 在数据库的主从设计中，从库的数据是通过同步主库的数据而来的。
◎ 计算机系统中主从驱动器的设计原理。

4.6.4　管道-过滤器模式

管道过滤器（Pipe-And-Filter）确定了系统的架构模式，它和工业流水线一样，原材料在通过流水线上的一道道工序后，形成了某种商业产品。在管道过滤器中，一个个的过滤器会对数据进行过滤，最后得到需要的数据。

管道负责数据的传递，它将原始数据传递给第 1 个过滤器，将一个过滤器的输出传递给下一个过滤器，作为下一个过滤器的输入，重复这个过程，直到处理结束。需要注意的是，管道只是对数据传输的抽象，它可能是管道，也可能是其他通信方式，甚至什么都没有（所有过滤器都在原始数据的基础上进行处理）。

过滤器只负责处理数据，可以有多个，每个过滤器都对原始数据做特定的处理，它们之间没有任何依赖关系，一个过滤器不必知道其他过滤器的存在。这种松耦合的设计，使得过滤器只需要实现单一的功能来降低了系统的复杂度，也使得过滤器之间的依赖最小，从而以更加灵活的组合来实现新的功能，如图 4.10 所示。

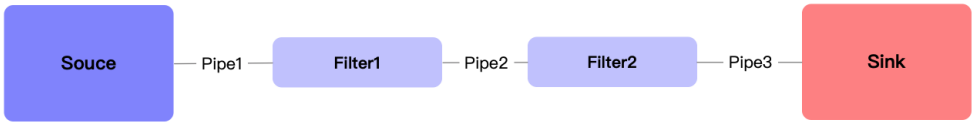


图 4.10

管道-过滤器模式的使用场景如下。

◎ 编译器。
◎ 生物信息学的工作流。

4.6.5　代理模式

代理模式用于在结构化系统中对组件进行解耦，系统内各组件之间采用远程调用的方式进行交互，代理组件充当组件间通信的协调角色，提供服务的组件将其能力（服务及特性）发布给代理，客户端均向代理请求服务，由代理将请求重定向到先前已发布过对应服务的组件进行处理，如图 4.11 所示。

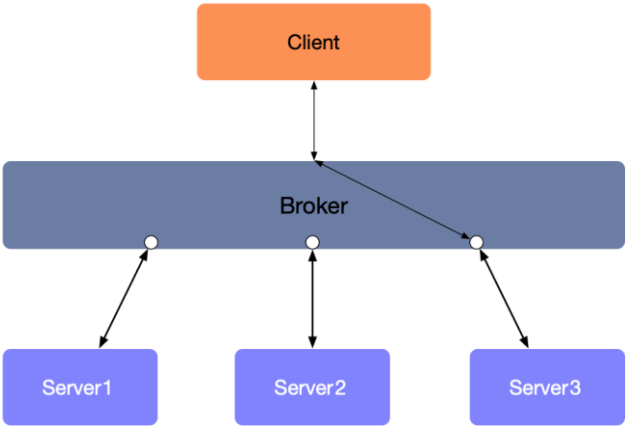


图 4.11

代理模式多用于消息代理软件中，例如 ActiveMQ、Kafka、RabbitMQ 和 JBoss Messaging 等。

4.6.6 P2P 模式

P2P（Peer-to-Peer，点对点）是一种网络技术，依赖网络中各节点的计算能力和带宽，而不是把依赖都聚集在有限的几台服务器上。P2P 架构的核心思想是每个节点既是客户端（Client），又是服务器端（Server），如图 4.12 所示。

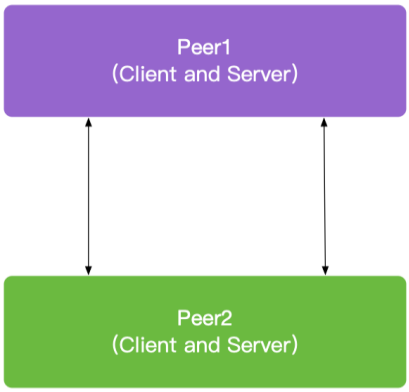


图 4.12

P2P（Point to Point，点对点）是下载术语，指我们的电脑在下载的同时要继续做主机上传。在这种下载方式中，参与下载的电脑越多，下载速度越快，但是对硬盘损伤较大（在写的同时还要读），对内存占用也较多，影响整机速度。

P2P 在网络隐私要求高和文件共享的领域得到了广泛应用，使用纯 P2P 技术的网络系统有比特币等。另外，P2P 技术也被用于类似 VoIP 等实时媒体业务的数据通信中。有些网络涉及搜索的一些功能，也使用客户端-服务器结构，但使用 P2P 结构来实现另外一些功能，这种网络设计模型不同于客户端-服务器模型，在客户端-服务器模型中通信通常来往于一台中央服务器。

P2P 网络主要用于让所有客户端都能提供资源，比如带宽、存储空间和计算能力。当有节点加入且对系统请求增多时，整个系统的容量也增加，这是具有一组固定服务器的客户端-服务器结构不能实现的，因为在这种结构中，客户端的增加意味着所有用户更慢速度的数据传输。P2P 网络的分布特性通过在多节点上复制数据展现，这也增加了防故障的健壮性，并且在纯 P2P 网络中，节点不需要依靠一个中心索引服务器来发现数据，系统也不会出现单点崩溃的情况。

P2P 模式的使用场景如下。

◎ 像 Gnutella 和 G2 这样的文件共享网络。
◎ 多媒体协议如 P2PTV 和 PDTP。
◎ 像 Spotify 这样的专有多媒体应用程序。

4.6.7　事件总线模式

事件总线模式如图 4.13 所示。这种模式主要处理事件，包括 4 个主要组件：事件源、事件监听器、通道和事件总线。事件源将消息发布到事件总线的特定通道上；事件监听器订阅特定的通道，它会被通知消息，这些消息被发布到它们之前订阅的一个通道上。

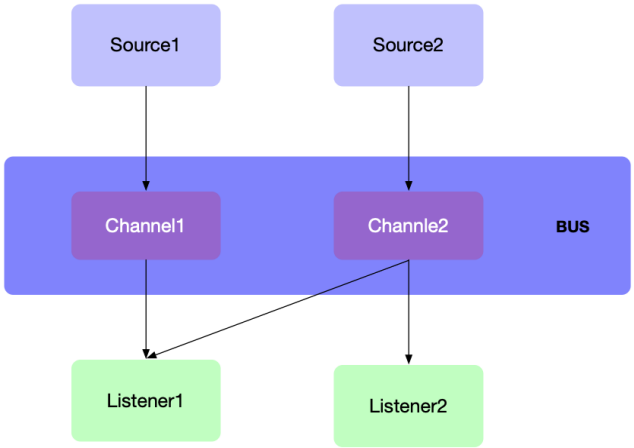


图 4.13

事件总线模式的使用场景如下。

◎ 安卓开发。
◎ 通知服务。

4.6.8 MVC 模式

MVC（Model View Controller，模型-视图-控制器）的概念存在很长时间了，但是随着互联网的发展，对客户端-服务器应用的需求日益增加，它才作为最佳设计方式被更多的人熟知。所有 Web 框架都基于 MVC 概念构建而成。MVC 设计模式的概念也非常容易理解。

◎ 模型（M）是数据的表述，它不是真正的数据，而是数据的接口。我们在使用模型从数据库中获取数据时，无须知道底层数据库错综复杂的知识。模型通常还会为数据库提供一层抽象，这样同一模型就能使用不同的数据库。
◎ 视图（V）是我们看到的界面，它是模型的表现层。在电脑中，视图是我们在浏览器中看到的 Web 应用的页面，或者桌面应用的 UI。视图还提供了收集用户输入的接口。
◎ 控制器（C）控制模型和视图之间的信息流动。它通过程序逻辑来判断模型从数据库中获取了什么信息，以及将什么信息传给了视图。它还通过视图从用户那里收集信息，并且实现业务逻辑、变更视图，或者通过模型修改数据，抑或二者兼具。

MVC 模式如图 4.14 所示。

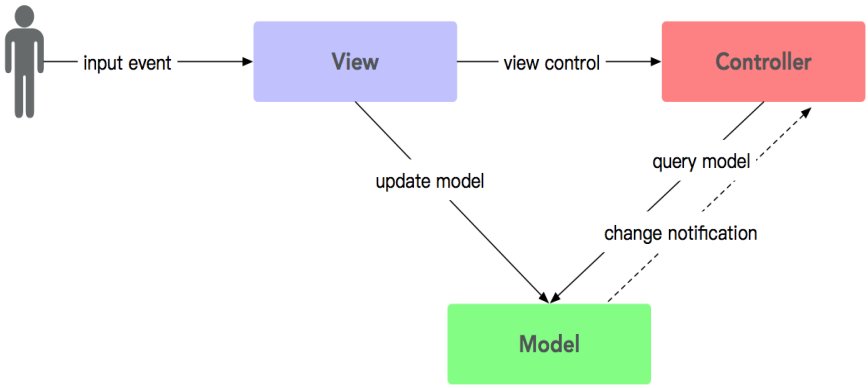


图 4.14

MVC 模式的使用场景如下。

◎ 互联网应用程序的体系架构。
◎ 像 Django 和 Rails 这样的 Web 框架。

4.6.9　黑板模式

黑板模式是一种常用的架构模式，是对观察者模式的扩展，由 3 个主要部分组成，如图 4.15 所示。

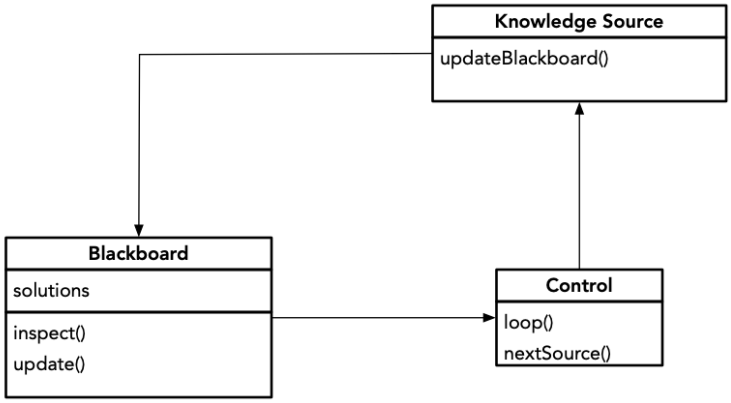


图 4.15

◎ 知识源：包含与应用程序相关的独立信息，信息源之间不直接进行通信，它们之间的交互只通过黑板完成。
◎ 黑板数据：黑板数据指按照与应用程序相关的层次来组织并解决问题的数据，知识源通过不断地改变黑板数据来解决问题。
◎ 控制：完全由黑板的状态驱动，黑板状态的改变决定了我们需要使用的特定知识。

所有组件都可以访问黑板。组件可以生成添加到黑板上的新数据对象，它在黑板上查找特定类型的数据，并通过与现有知识源的模式匹配来查找这些数据。

黑板模式的使用场景如下。

◎ 语音识别。
◎ 车辆识别和跟踪。
◎ 蛋白质结构识别。
◎ 对声呐信号的解释。

4.6.10 解释器模式

解释器模式属于行为型模式，实现了一个表达式接口，该接口解释一个特定的上下文，常用于符号处理引擎、SQL 解析等中。解释器模式如图 4.16 所示。

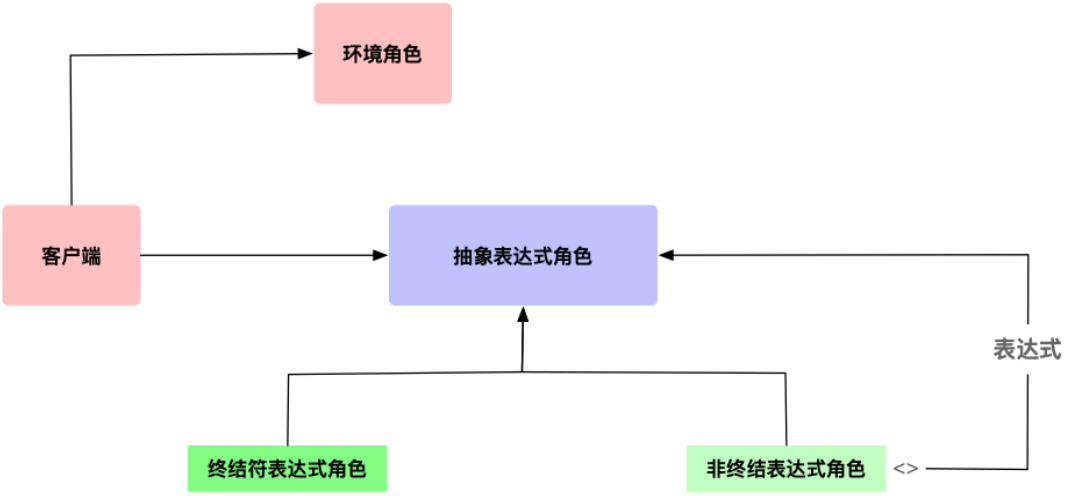


图 4.16

下面对如图 4.16 所示的名词进行解释。

◎ 抽象表达式角色（Expression）：声明所有的具体表达式角色都需要实现的一个抽象接口。这个接口主要是一个 interpret()方法，叫作解释操作。
◎ 终结符表达式角色（Terminal Expression）：实现了抽象表达式角色所要求的接口，主要是一个 interpret()方法；文法中的每一个终结符都有一个具体的终结表达式与之对应。比如有一个简单的公式 R=R1+R2，其中的 R1 和 R2 就是终结符，对应的解析 R1 和 R2 的解释器就是终结符表达式。
◎ 非终结表达式角色（Nonterminal Expression）：文法中的每一条规则都需要一个具体的非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字，比如在公式 R=R1+R2 中，“+”就是非终结符，解析“+”的解释器就是一个非终结符表达式。
◎ 环境角色（Context）：这个角色的任务一般是存放文法中各个终结符所对应的具体值，比如 R=R1+R2，我们给 R1 赋值 100，给 R2 赋值 200，这些信息都需要被存放到环境角色中，在很多情况下我们使用 Map 来充当环境角色就够了。

解释器模式的使用场景如下。

◎ 数据库查询语言，比如 SQL。
◎ 用于描述通信协议的语言。

4.7　抽象

抽象是架构思维里面的一个重点，其中包括两个层面的内容：一个是常说的归纳方法，即各种类似场景的实现太多了，自然就归纳了一种规则或方法出来供以后的设计用；另一个是抽象，它更加重要，即将非类似场景中的共性内容也总结出来，进一步抽象为类似的东西，以更加方便地适应各种变化。

4.8　结构化

结构化是一种从框架到细节的思维方式，强调在分析问题的过程中，不先入为主，不马上陷入细节。结构化也被称为表格化的思维结构，强调结构和逻辑，通过结构的完整和逻辑的严密来保证结果的正确。结构化的核心在于对问题进行正确界定的基础上（以终为

始），对问题的构成要素进行合理分类，并对其中的重点环节进行分析（要事第一）。

4.8.1 结构化的原则

结构化的原则如下。

◎ 以终为始。
◎ 知道做这件事情的目标是什么。
◎ 根据这个目标倒推需要完成哪些工作和任务。
◎ 做任何一件事情都必须有一个目标，才能在分析论证过程中得到预期的结果。
◎ 不要先入为主，避免陷入细节。
◎ 必须对各种观点进行分组，分组后的思想观点在经过不同层次的抽象后构成金字塔。
◎ 分类原则 MECE（Mutually Exclusive Collectively Exhaustive，相互独立、完全穷尽）。
◎ 相互之间具有排他性，整体而言毫无遗漏。

简单地说，结构化的原则就是不重叠、不遗漏。

4.8.2 结构化分析工具

结构化工具主要有如下几种。

◎ 逻辑树：又叫作问题树、演绎树或分解树等，是将所有问题的子问题都分层罗列，从最高层开始逐步向下扩展。
◎ 议题树：将问题分解成多个利于操作的小块并分别进行处理，在解决问题的早期还没有足够形成假设的基础。
◎ 假设树：先给出解决问题的假设方案，再举出所需的充足原因来验证或推翻该假设，较早集中于潜在的问题，加快解决问题的进程，对情况有足够的了解，能提出合理的假设方案。
◎ 是否树：列出关键问题，使用“是”和“否”来回答，按照需要采取相应行动的逻辑顺序来排列，确认对目前要做的决定有关键意义的问题，应该对事务及其结构有良好的理解，并可作为沟通工具。
◎ 二维表格。

◎ 80/20 法则：80/20 法则是分析式的方法。80/20 分析法用于让人注意到造成某种状况的关键原因，也就是找到那些导致 80%产出的 20%投入。

4.8.3　结构化思维的 7 个环节

结构化思维的 7 个环节如下。

◎ Who：谁来做？
◎ What：内容是什么、做什么？
◎ When：什么时候做？
◎ Where：在哪里做？
◎ Why：为什么做这件事情？
◎ How：怎么做？
◎ How Much：用多少时间和其他资源？

4.9　迭代

迭代思维是我们在架构思考中需要考虑的另一个内容。没有最优的架构，只有不断进化的架构和最适合的架构，因此架构本身也在随着业务需求的变化不断迭代和演化，而不是追求用最新的技术一步到位。

架构设计往往发生在需求的细节尚未完成时。因此，随着项目的进行，需求还可能细化、变更。原先的架构肯定会有不足或错误的地方。

借用一句名言，“凡事预则立，不预则废”，在软件设计初期，投入精力进行架构设计是很有必要的，这个架构是我们在后续的设计、编码过程中依赖的基础。我们应用迭代方法的最大目的就是稳步地改进软件架构。

软件架构的改进在软件开发过程中会经历一个振荡期，这个振荡期可能横跨了数个迭代周期，其间的架构设计将会经历剧烈的变化，但最后一定会趋向于平稳。

4.10　勿做过度设计

如果架构师创造的功能无法正好迎合需求或实现业务目标或不必要的特征，则会出现

过度设计的现象，这一般是因为不清楚项目需求或不确定架构的性能。为了避免出现过度设计的现象，我们需要使需求透明化。过度设计的迹象并不总是显而易见的，但识别这些迹象对节省时间和降低成本非常重要。不需要做超出需求的设计，不需要假设不存在的极端条件做系统架构设计。

过度设计的特征如下。

◎ 超高强度。
◎ 终极性能。
◎ 重新设计。
◎ 过度合理。

过度设计的影响如下。

◎ 开发时间和成本。每个新功能都必须经过设计、研发、测试的步骤。这样大大增加了人力和时间成本，导致项目的交付时间延长。
◎ 维护成本。过于复杂的架构设计都会导致各种问题，增加了维护成本，并且让整个系统的维护过程变得复杂。

过度设计的原因如下。

◎ 需求透明度。如果在架构设计之初并未完全理解需求方提出的需求，或遗漏了关键的需求，出现误解的可能性就会大大提高，从而导致架构设计过度。
◎ 架构性能未知。高估了产品的市场预期。

第 5 章

架构设计

在架构设计过程中，我们会根据需要做出不同的架构设计，而在设计时需要涉及一定的架构设计核心要素。

5.1 架构设计概要

架构设计是从业务需求到系统实现的一个转换，是对需求进一步深入分析的过程，用于确定系统中实体与实体的关系，以及实体的形式与功能。架构可根据从业务需求到系统实现的不同需要分为：业务架构、应用架构、数据架构、技术架构。下面以电商系统为例进行架构设计。

5.1.1 业务架构

业务架构是对业务需求的提炼和抽象，使用一套方法论对产品（项目）所涉及需求的业务进行业务边界划分，简单地讲就是根据一套逻辑思路进行业务的拆分，开发软件必须满足业务需求，否则就是空中楼阁。软件系统在业务上的复杂度问题，可以从业务架构的角度切分工作交界面来解决。比如在做一个电商网站时，需要将商品类目、商品、订单、支付、退款等功能很清晰地划分出来，不要在业务架构中考虑用什么技术开发、并发量是否太大、选择什么样的硬件，等等。

这里简单规划了电商系统的业务模块，对其根据业务架构的模块不断细化，一直分解到代码流程。对于软件开发而言，以业务架构为依托，架构师和开发者能比较清晰地看到系统的业务全貌，架构师能更方便地分析系统复杂度，分解业务逻辑，做好开发的分工，如图 5.1 所示。



图 5.1

业务架构是决定一个软件项目能否顺利开展的总纲，软件架构是业务架构在技术层面的映射，合理的开发分工也应该基于业务架构去做。如果没有业务架构，进行软件开发就会很盲目。业务架构是需求和开发的汇聚点，需求分析是否做到位，功能开发是否达到预期目标，都以此为依托。我们在工作中会遇到一些问题，例如研发人员说需求分析做得不到位，而做需求的人员会质疑需求做到怎样才算到位，为什么开发出的产品和用户想要的不一致，这些从根上来说，都是因为没有将业务架构梳理清楚，没有达成共识。

站在软件项目的角度来看，在项目前期做好业务架构设计，对整个项目的开发都有重要的意义。例如，对于比较类似的业务系统，可能业务架构在比较粗的颗粒度上是一样的，而在细化过程中不一样。在做项目时，当手头有一个现成的系统，需要做一个需求类似的项目时，大家可能会首先尝试用现成的系统去覆盖新项目，以求利益最大化。对于该想法能否实现，可以通过业务架构来衡量，如果没有业务架构，则接下来的工作会非常盲目。

业务架构的设计原则如下。

（1）将业务平台化。

◎ 业务平台化，相互独立，例如交易平台、物流平台、支付平台、广告平台等。
◎ 基础业务下沉，可复用，例如用户、商品、类目、促销、时效等。

（2）将核心业务和非核心业务分离。将电商系统的核心业务和非核心业务如主交易服务和通用交易服务分离，将核心业务精简（利于稳定），并将非核心业务多样化。

（3）隔离不同类型的业务。

◎ 交易平台的作用是让买家和卖家签订交易合同，所以需要优先保证高可用，让用户能快速下单。
◎ 履约业务对可用性没有太高要求，但要优先保证一致性。
◎ 秒杀业务对高并发要求很高，应该和常规业务分离。

（4）区分主流程和辅助流程。要清楚哪些是电商系统的主流程，在运行时优先保证主流程的顺利完成；对辅助流程可以采用后台异步的方式，避免辅助流程的失败影响主流程的失败回流。

5.1.2　应用架构

应用架构介于业务与技术之间，是对整个系统实现的总体架构，需要指出系统的层次、系统开发的原则、系统各个层次的应用服务。如图 5.2 所示为将系统分为表现层、业务流程层、服务层、服务构建层、数据层、集成层、数据架构层和服务治理层，并写明每个层次的应用提供的服务。

在进行系统拆分时，要平衡业务和技术的复杂度，保证系统形散神不散。系统采用什么样的应用架构，则受到业务复杂度的影响，包括企业的发展阶段和业务特点；同时受技术复杂度的影响，包括 IT 技术的发展阶段和内部技术人员的水平。业务的复杂度（包括业务量大）必然带来技术的复杂度，应用架构的目标是在解决业务复杂度的同时避免技术太复杂，确保业务架构落地。

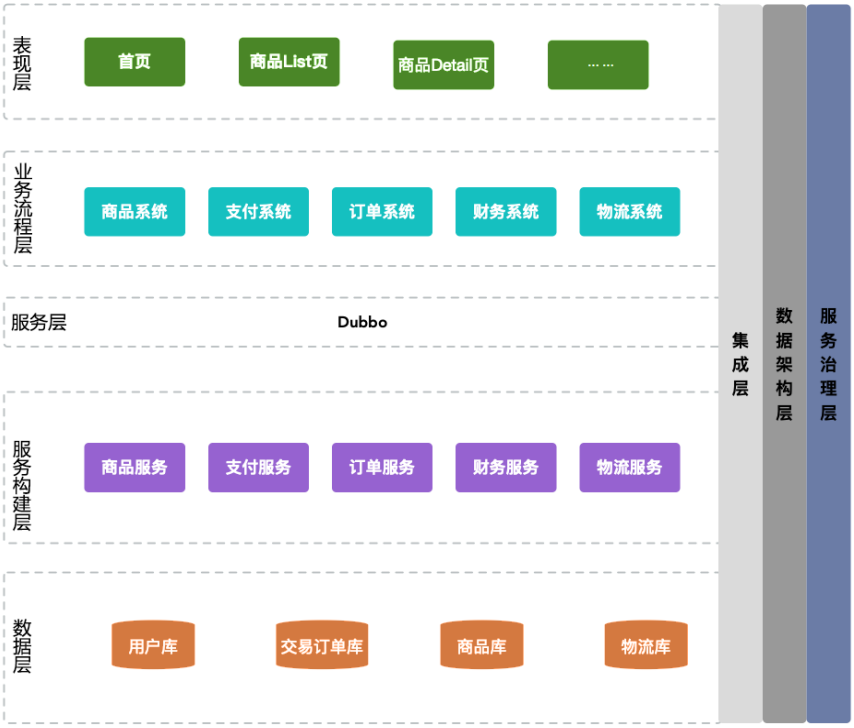


图 5.2

应用架构的设计原则如下。

（1）稳定

◎ 一切以稳定为中心。
◎ 架构尽可能简单、清晰，追求小而美，不要大而全。
◎ 不过度设计。

（2）解耦

◎ 将稳定部分与易变部分分离。
◎ 将核心业务与非核心业务分离。
◎ 将电商主流程和辅助流程分离。
◎ 将应用与数据分离。
◎ 将服务和实现细节分离。

（3）抽象

◎ 应用抽象化：应用只依赖服务抽象，不依赖服务实现的细节和位置。
◎ 数据库抽象化：应用只依赖逻辑数据库，不需要关心物理库的位置和分片。
◎ 服务抽象化：应用虚拟化部署，不需要关心实体机的配置，动态调配资源。

（4）松耦合

◎ 跨域调用异步化：在不同的业务域之间尽量异步解耦。
◎ 非核心业务尽量异步化：在核心业务和非核心业务之间尽量异步化。
◎ 在必须同步调用时，需要设置超时时间和任务队列的长度。

（5）容错设计

◎ 服务自治：服务能彼此独立修改、部署、发布和管理，避免引发连锁反应。
◎ 集群容错：应用系统集群部署，避免单点服务。
◎ 多机房容灾：多机房部署、多活。

5.1.3　技术架构

技术架构就是对在业务架构中提出的功能（或服务）进行技术方案的实现，包括软件系统实现、操作系统选择和运行时设计。技术架构的边界比较模糊，对于不同的受众，内容的详细程度也不同，技术栈自上而下比较关注技术架构，但是各层关注的点不同。技术决策层可能关心的是系统或系统群的技术选型，对整体的把握要保证不因为选型引起其他风险，例如，如果在高性能存储方面选择 Redis，就要尽量保证网络的封闭性，避免公网访问；再如，在选择以 COBOL 语言实现的各类产品时，要考虑市场上开发人员数量少，需要承担更高的迭代成本等。

上述业务架构的一个简单技术架构如图 5.3 所示。

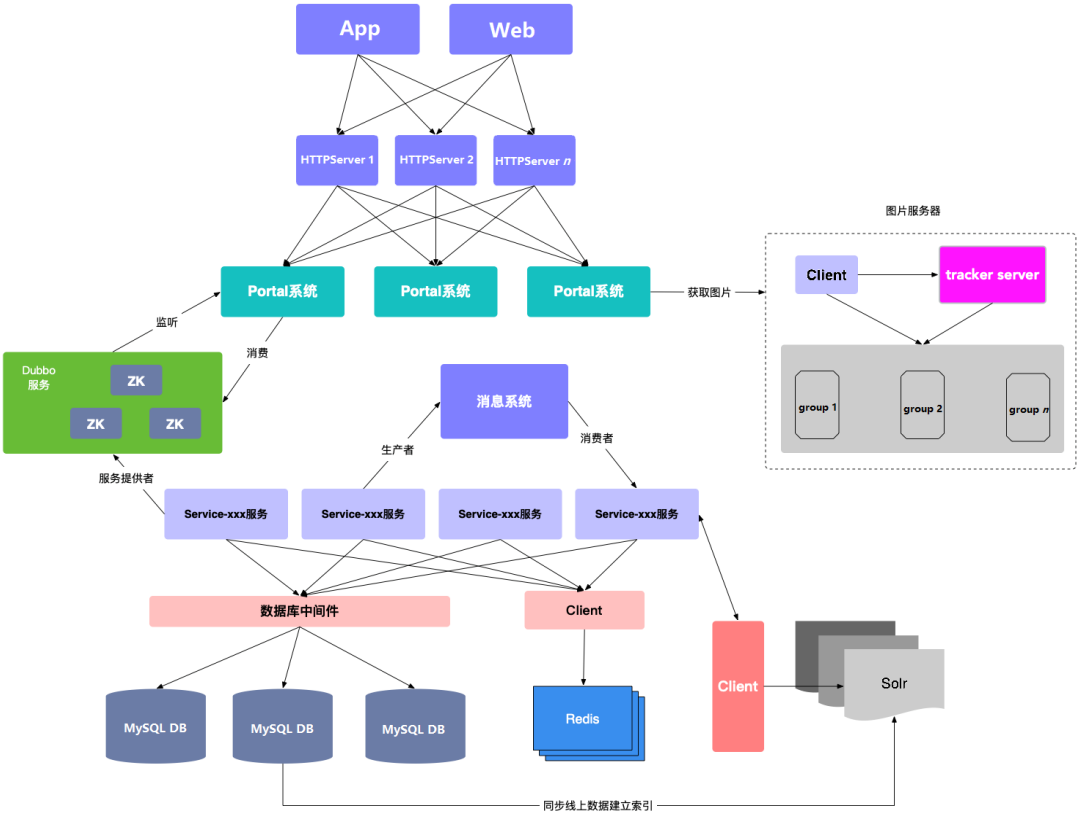


图 5.3

技术架构的设计原则如下。

（1）无状态，即尽量不要把状态数据保存在本机上。

（2）可复用。

◎ 复用粒度是有业务逻辑的抽象服务，不是服务的实现细节。
◎ 服务引用只依赖服务抽象。

（3）松耦合

◎ 跨业务域调用，尽可能异步解耦。
◎ 在同步调用时设置超时时间和队列大小。
◎ 将相对稳定的基础服务与易变流程服务分离。

（4）可治理

◎ 服务可降级。
◎ 服务可限流。
◎ 服务可开关。
◎ 服务可监控。
◎ 白名单机制。
◎ 制定服务契约。

（5）基础服务

◎ 基础服务下沉、可复用，例如时效、库存和价格计算。
◎ 基础服务自治、相对独立。
◎ 对基础服务的实现要精简，并可水平扩展。
◎ 对基础服务的实现要进行物理隔离，包括基础服务相关的数据。

5.1.4　数据架构

数据架构是对存储数据（资源）的架构，其设计原则和应用架构设计大同小异，在设计时需要考虑系统的业务场景，需要根据不同的业务场景对数据进行异构设计、数据库读写分离、分布式数据存储策略等。如图 5.4 所示是电商系统中数据架构的一个概要。

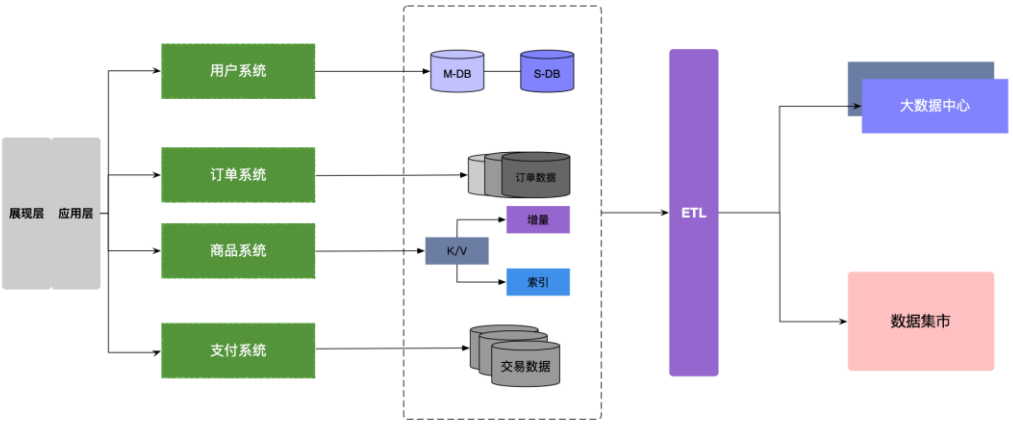


图 5.4

数据架构包括两部分内容：静态部分的内容和动态部分的内容。静态部分的内容的重点是数据元模型、数据模型，包括主数据、共享动态数据和所有业务相关的业务对象数据的分析和建模；动态部分的内容的重点则是对数据全生命周期的管控和治理。因此，不能单纯地将数据架构理解为纯静态的数据模型。业务架构中数据模型的分析重点是主数据和核心业务对象，应用架构中数据模型的分析重点则进一步转换为逻辑模型和物理模型，直到最终的数据存储和分布。

数据分两个层面的生命周期：单业务对象数据的全生命周期，它往往和流程建模中的单个工作流或审批流相关；跨多个业务域对象数据的全生命周期，它体现的是多个业务对象数据之间的转换和映射，往往和端到端的业务流程 BPM 相关。这里要注意，数据虽然是静态层面的内容，但是数据的生命周期或端到端的数据映射往往间接反映了流程，这是很重要的内容。

数据建模的方法包括面向结构的传统 ER 模型分析方法，也包括面向对象的对象类模型分析方法，它们都是可行的数据建模方法，只是传统 ER 模型分析方法更容易实现向底层物理数据库模型的转换，而面向对象的对象类建模方法更容易体现抽象和复用。特别是在企业架构建模中，面向对象和面向结构往往不是严格区分的，很多时候都会出现两种方法混用的情况，但重点是区分每种方法或工具的重点及要解决的问题。

与数据相关的矩阵分析相当多，业务架构阶段的重点矩阵分析是业务对象和业务流程、业务组件、业务功能间的类 CRUD 矩阵分析；而应用架构阶段的重点矩阵分析是逻辑或物理模型对象和具体的应用模块或应用功能间的矩阵分析。两者的思路基本类似，只是关注的层面不同，前者重点关注主数据的识别和业务组件的分析，而后者重点关注应用功能模块的划分和模块间集成接口的初步分析。

根据前面的思路，我们仍然应该将数据集成分析分解为两个层面的内容：业务层面的分析，以及应用和 IT 实现层面的分析。前者的重点是理清业务流程或业务域之间的业务对象集成和交互，而后者的重点是如何更好地共享数据或如何通过类似的 BI 工具或大数据平台实现数据的集成和交互。

数据架构的设计原则如下。

（1）统一数据视图，即保证数据的及时性、一致性、准确性和完整性。

（2）数据和应用分离。

◎ 应用系统只依赖逻辑数据库。

◎ 应用系统不直接访问其他应用的数据库，只能通过接口访问。

（3）数据异构，即在源数据和目标数据内容相同时做索引异构，在商品库不同维度的内容不同时（如订单数据中的买家库和卖家库）做数据库异构。

（4）数据库读写分离。

◎ 将访问量大的数据库做读写分离，例如订单库。
◎ 将数据量大的数据库做分库分表。
◎ 将不同业务域的数据库做分区隔离。
◎ 对重要的数据配置备库。

（5）采用关系数据库。除成本因素外，MySQL 的数据库扩展性和高并发支持能力较强，也比较容易招聘到相关的研发人员和运维人员。

（6）合理利用 NoSQL 数据库。在数据库有能力支撑时，尽量不要引入缓存。另外，要合理利用缓存做容灾。

5.2　架构设计的流程

所有软件开发方法都要解决从需求到实践的转换问题，为了提高软件的质量，前辈们提出了分析工程和各种建模技术的需求，但是需求和设计之间的鸿沟还是很难逾越的，也就是说，缺乏从需求到设计进行衔接并做决策的中间过程，于是软件架构设计应运而生。

人们已经对架构设计提出了许多方法：工件驱动的方法、用例驱动的方法、模式驱动的方法、领域驱动的方法。这里展示一种经典的架构设计过程模型，它沿用了 RUP 中迭代增量的思想，由分析、描述、选择、构造和组合 5 个阶段组成，如图 5.5 所示。

有兴趣的读者可以学习架构设计的元模型，来设计属于自己领域或者产品线的设计过程模型，其实下面的模型也是元模型的实例化。

如图 5.5 所示，系统依据需求规格说明书分析出功能需求和架构需求，通过用例和场景的描述，将需求分为关键的、次要的和可选的 3 类。关键需求决定架构，结合软件架构风格和通用知识选择最关键、影响最大的子系统分析设计并产生构件。组合就是定义构件接口，构件作为一个封闭的功能实体对外提供交互接口，并通过连接件将构件连接起来形成最终的软件架构描述。这是个不断迭代的过程，在每一次迭代中，都选取并实现一组用例和场景来确认并完善架构。

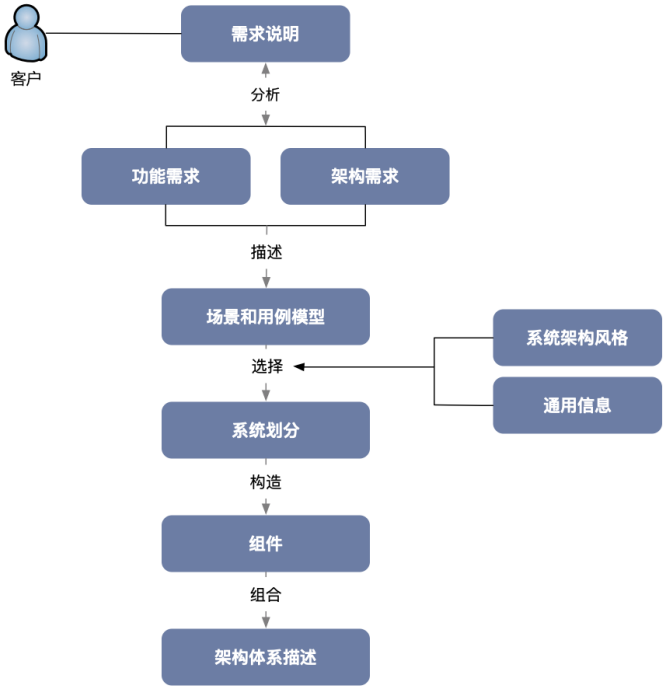


图 5.5

这个过程模型看似很流畅，架构师在设计时却很难把握它的正确性和精准性，而且用它构架的系统是否对后续设计开发形成一种原则上的指导是很难说的。我们知道，层次化分析是解决复杂问题的一般性方法，下面的方法将从层次化的角度来设计系统的架构模型，它将直接指导开发人员实现系统，是一种架构设计的理想方式。

接着讲讲架构设计方法之四层驱动模型。在软件开发过程中是存在多个层次的，对于每个层次，驱动其进行架构设计的因素也有所不同，所以更好的方式是，区分和建立必要的层次，从而形成一种层次化的多因素驱动的软件架构设计模型。我们将整个软件架构的设计划分为 4 个层次：目标层、信息层、构件层和实现层。同时，软件的整体复杂性也通过这 4 个不同的抽象层次清晰地刻画出来。模型中的每一层都有一种因素驱动着建模设计的进行：目标层的驱动因素是所要实现系统的各种相关角色；信息层的驱动因素是目标层中的各种目标；构造层的驱动因素是信息层中的各种信息及其信息活动；实现层的驱动因素是构件层中的类、对象、对象交互等各种面向对象设计的元素，实现层所产生的接口、类及其属性、方法的具体语言代码及实现框架则对接下来的编码实现阶段直接提供支持。

5.3　架构设计的核心要素

一般来说，软件架构需要关注性能、可用性、伸缩性、扩展性和安全性这 5 个要素，下面分别对这 5 个要素进行讲解。

5.3.1　性能

性能是架构设计的一个重要方面，任何软件架构设计方案都必须考虑可能带来的性能问题。也正是因为性能问题几乎无处不在，所以优化网站性能的手段非常多。

◎ 浏览器端：可以使用浏览器缓存、页面压缩传输、合理布局页面、减少 Cookie 传输等手段，甚至可以使用 CDN 加速功能。
◎ 应用服务器端：可以使用服务器本地缓存和分布式缓存，也可以通过异步操作方式来加快响应。在高并发请求的情况下，可以将多台应用服务器组成一个集群共同对外服务，提高整体处理能力，改善性能。
◎ 数据库服务器端：可以使用索引、缓存、SQL 性能优化等手段，还可以使用 NoSQL 数据库来优化数据模型和存储结构等。
◎ 衡量架构性能的一系列指标：重要的指标有响应时间、TPS、系统性能计数器等，通过这些指标可以确定系统设计是否达标。

5.3.2　可用性

可用性即系统不间断地提供服务的时间。几乎所有网站都承诺 7×24 小时提供服务，但任何网站都不可能完全做到，总会有一些不可用的时间，扣除这些不可用的时间后，就是网站的可用时间。一些架构可以做到 4 个 9 以上的可用性，也就是 99.99%。

让架构高可用的主要手段是冗余，应用部署在多台服务器上同时提供服务，数据存储在多台服务器上相互备份，任何一台服务器都不会影响应用的整体可用，通常的实现手段是将多台服务器通过负载均衡设备组成一个集群。

衡量一个系统架构设计是否满足高可用的目标，就是假设系统中任何一台或者多台服务器宕机，以及出现各种不可预期的问题，而这时系统整体是否依然可用。

5.3.3 伸缩性

大型网站需要面对大量用户的高并发访问和海量数据存储，网站通过集群的方式将多台服务器组成一个整体共同提供服务。所谓的伸缩性是指通过不断向集群中加入服务器来缓解不断增加的整体上的用户并发访问压力和数据存储需求。

衡量架构伸缩性的主要标准就是能否用多台服务器构建集群，是否容易向集群中添加新的服务器，加入新的服务器后是否可以提供和原来的服务器无差别的服务，以及集群可容纳的总服务器数量是否有限制。

5.3.4 扩展性

不同于其他架构要素主要关注非功能性需求，网站的扩展性架构直接关注网站的功能需求。网站在快速发展，功能也在不断扩展，能够快速响应需求的变化，是设计网站可扩展架构的主要目标。

衡量网站架构扩展性好坏的主要标准就是在网站增加新的业务产品时，是否可以实现对现有产品透明、无影响，不同产品之间是否很少耦合等。网站可扩展架构的主要手段是事件驱动架构和分布式服务。事件驱动通常利用消息队列实现，通过这种方式将消息生产和处理逻辑分离。

服务器服务则是将业务和可复用服务分离，通过分布式服务框架进行调用。在新增产品时，可以通过调用可复用的服务来实现自身的业务逻辑，对现有产品没有任何影响。

5.3.5 安全性

互联网是开放的，网站的安全架构就是保护网站不受恶意访问和攻击，避免网站的重要数据被窃取。衡量网站安全架构的标准就是针对现存和潜在的各种攻击和窃密手段，是否有可靠的应对策略。

5.4 高性能设计

下面从用户、开发者和运维人员的视角，来看看他们关注的性能是什么。

从用户的视角来看，性能相当于从提交请求到看到页面的时间。不同计算机的性能差异，不同浏览器解析 HTML 的速度差异，以及不同网络提供商提供的互联网服务的速度差异，等等，都会导致从提交请求到看到页面的时间远大于服务器处理请求的时间。在实践中，我们可以采用一些前端架构优化手段，例如优化 HTML 样式，利用浏览器异步和并发的特性，调整缓存策略，使用 CDN 服务、反向代理等，使用户尽快看到内容，即使不对应用服务进行优化，也能够很好地改善用户的体验。

从开发者的视角来看，性能的重点在于应用服务器的性能，包括响应延迟、系统吞吐量、并发处理、系统稳定性等，主要的优化手段是利用缓存加速数据读取，使用集群提高吞吐量，使用异步加快请求响应，优化代码改善程序，等等。

从运维人员的视角来看，性能相当于一些基础设施的性能和利用率。

5.4.1　主要的性能测试指标

主要的性能测试指标有响应时间、并发数、吞吐量、性能计数器等，下面分别进行讲解。

◎ 响应时间：指从发出这个请求开始到接收到数据的时间，在一般情况下，这个时间都非常短，甚至不超过测试的误差值，所以我们可以采用重复请求的方式来获取具体的响应时间，比如请求 10 万次，并记录总时间，然后计算出单次请求的时间。

◎ 并发数指：指能够同时处理的请求数目，对于网站而言就是并发用户数。其中，网站系统用户数 > 网站在线用户数 > 网站并发用户数。

◎ 吞吐量：指在单位时间内能够处理的请求数，体现了系统的整体处理能力。衡量指标有很多，可以是每秒请求数、每秒页面数、每天的访问人数及每小时处理的业务数等。常用的量化指标有 TPS（每秒事务数）、HPS（每秒 HTTP 请求数）、QPS（每秒查询数）等。

◎ 性能计数器：指描述服务器或操作系统的一些性能指标，包括系统负载（System Load）、线程数、内存使用、磁盘和网络 I/O 等，当这些值超过警告值（安全临界值）时，就会向开发运维人员告警，通知其及时处理异常。

5.4.2 性能测试方法

性能测试是一个统称，具体可以分为性能测试、负载测试、压力测试和稳定性测试。

◎ 性能测试：指以初期设计的指标为预期目标，不断对系统施压，看系统在预期的范围内能否达到预期的性能。
◎ 负载测试：指对系统不断增加并发请求以增加系统压力，直到系统的某项或多项指标达到安全临界值，这时继续对系统施加压力，系统的处理能力会有所下降。
◎ 压力测试：指在超过安全负载的情况下继续施压，直到系统崩溃或不再能够处理任意请求，以此来计算系统能承受的最大压力。
◎ 稳定性测试：指在一定的压力（不均匀施压）下，系统能够稳定运行较长时间。

如图 5.6 所示，a 到 b 区间就是网站日常的运行区间，服务处理大多在这一区间内；c 点相当于系统的最大负载点；b 到 c 区间指访问量因某些原因超过了日常访问压力；超过了 c 点后，继续增加压力，这时系统的性能开始下降，但是资源消耗会更多，直到 d 点（系统的崩溃点），超过 d 点继续加压的话，系统将不能处理任何请求。

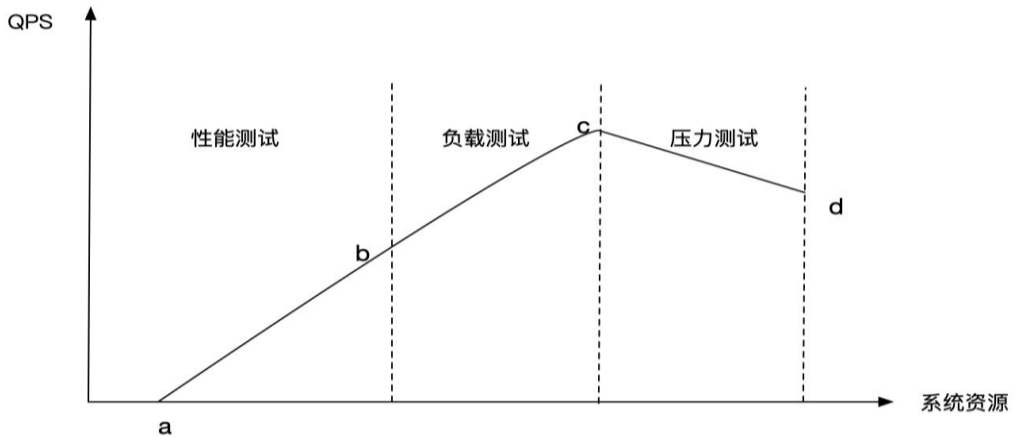


图 5.6

性能测试反映的是系统的处理能力，与其对应的是用户的等待时间（响应时间），如图 5.7 所示。

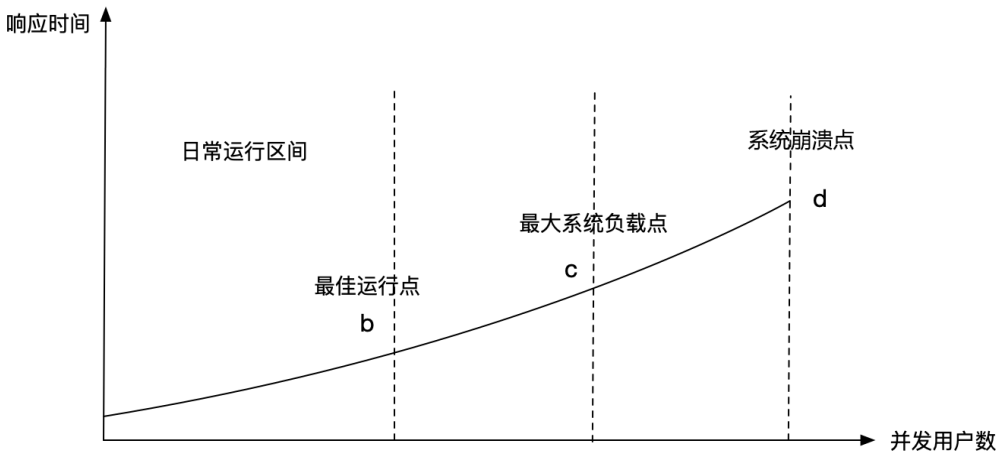


图 5.7

图 5.7 中的点与图 5.6 中的点相互对应，直到系统崩溃，用户失去响应。

5.4.3　性能优化策略

根据系统分层架构，性能优化可分为 Web 前端性能优化、应用服务器性能优化和存储性能优化。

1. Web 前端性能优化

对于 Web 前端，可以从浏览器访问、CDN、代理服务器这三方面进行性能优化。

1）浏览器访问优化

浏览器访问优化的主要方法如下。

◎ 减少 HTTP 请求数，主要通过合并 CSS、JavaScript 和图片实现。
◎ 使用浏览器端缓存。在某些时候，对静态资源文件的编写需要及时应用到客户端浏览器，可通过改变文件名实现。
◎ 启用页面压缩，文本文件的压缩效率可达 80%以上。
◎ 将 CSS 放在页面最上面，将 JavaScript 放在页面最下面。
◎ 减少 Cookie 传输，可以考虑通过独立域名来传输 Cookie。

2）CDN

CDN（内容分发网络）在本质上也属于缓存，指将数据缓存在离用户近的地方，使用户尽快获取数据，因为 CDN 都部署在网络运营商的机房中，这些运营商同时为用户提供网络服务，因此用户请求的路由会优先到达 CDN 服务器，如果存在请求的资源，就直接返回，以最短路径返回响应，加快用户的访问速度，还能够为中心机房减轻压力。

CDN 一般用来缓存静态资源、CSS、JavaScript 脚本、静态页面、图片等，这些内容的修改频率很低但是访问请求频率很高，因此放在 CDN 上能够很好地改善访问速度。

3）代理服务器

传统的代理服务器是当用户请求不到想要的资源时，由代理服务器帮用户请求，用户知道自己请求的最终服务器是谁，比如 VPN 通过代理服务器将请求转发到站外服务网络。而反向代理是当用户请求一个地址时，用户请求的是反向代理服务器，然后由反向代理服务器请求其他服务器获取内容，用户不知道最终是从哪一台服务器获取的数据。

反向代理 Web 服务器接收 HTTP 请求，然后进行请求转发，在获取内容后将内容返回给用户，用户只知道是由反向代理服务器给自己的数据，而不知道数据源最终是从哪个服务器来的。反向代理服务器具有保护作用，来自互联网的请求都需要经过反向代理服务器，相当于在 Web 服务器之间建立起一道屏障。

除安全外，可以在反向代理服务器上进行一些静态资源的缓存，以此来加快访问速度，减轻应用服务器的负载压力。

当然，也可以将某些动态资源缓存在代理服务器上，比如热门的词条、帖子、博客等，这些资源的请求量可能非常大，如果每次都走一遍流程的话，就会带来很大的压力。同时，当这些动态内容发生改变时，会通知反向代理服务器缓存失效，代理服务器会重新缓存动态资源。

除此之外，反向代理服务器还可以用来做负载均衡，通过负载均衡来构建服务器集群，以此来提高系统的总体处理能力，进而提高应用服务器处理高并发的能力。

2. 应用服务器的性能优化

应用服务器就是处理网站业务的服务器，网站的业务代码都部署在这里，主要的优化手段有缓存、集群、异步等。

3. 存储性能优化

可以考虑使用分布式存储、磁盘阵列和 HDFS（Hadoop）。

5.5　高可用设计

高可用性（High Availability）通常用来描述一个系统经过专门的设计，减少了停工时间，并保证了其服务的高度可用性，简而言之，就是不间断地对外提供服务。

5.5.1　高可用性的度量与考核

对高可用性度量的公式为：

◎ 网站不可用时间（故障时间） = 故障修复时间点 − 故障发现（报告）时间点；
◎ 网站年度可用性指标 =（1 − 网站不可用时间/年度总时间）× 100%。

业界通常用多个 9 来衡量网站的可用性，如下所述。

◎ 2 个 9：表示基本可用，不可用时间少于 88 小时。
◎ 3 个 9：表示较高可用，不可用时间少于 9 小时。
◎ 4 个 9：表示具有自动恢复能力的高可用，不可用时间少于 53 分钟。
◎ 5 个 9：表示极度高可用，不可用时间少于 5 分钟。

可用性指标是网站架构设计的重要指标，对外是服务承诺，对内是考核指标。从管理层面来看，可用性指标是网站或系统架构的整体考核指标，具体到每个工程师的考核，更多地使用故障分。故障分是指对网站故障进行分类加权计算故障责任的方法，如表 5.1 所示。

表 5.1

分　类	描　述	权　重
事故级故障	严重故障，网站整体不可用	100
A 类故障	网站访问不顺畅或核心功能不可用	20
B 类故障	非核心功能不可用，或核心功能少数用户不能访问	5
C 类故障	其他故障	1

故障分的计算公式：故障分 = 故障时间（分钟）× 故障权重。

5.5.2 高可用的架构

典型的网站设计遵循基本分层架构模型：应用层（处理具体的业务逻辑）、服务层（提供可复用的服务）、数据层（数据的存储和访问）。

1. 高可用的应用层

应用层主要处理业务逻辑，一个显著特点是应用的无状态性。

所谓无状态是指应用服务器不保存业务的上下文信息，而仅根据每次请求提交的数据进行相应的业务逻辑处理，多个服务实例（服务器）之间完全对等，请求被提交到任意服务器后，处理结果都是完全一样的。

高可用的应用层的第 1 个特点是通过负载均衡实现失效转移。

在业务量和数据量都较高的情况下，通过负载均衡，可以将流量和数据都分摊到一个集群所组成的多台服务器上，以提高整体的处理能力。目前的负载均衡软、硬件都提供了失效转移功能。所以，如果集群的服务是无状态的对等服务，则负载均衡技术可以保证系统的高可用性。

当 Web 服务器集群中的服务器都可用时，负载均衡服务器会将访问请求分发给任意一台服务器进行处理。负载均衡服务器会通过心跳检测机制，在发现某台服务器（如图 5.8 所示的 Web 服务器-001）失去响应时，将该服务器从服务器列表中剔除。为了保证高可用性，即使某个应用的访问量非常少，也必须部署至少两台服务器，通过负载均衡构建一个小型集群。

可以看出，高可用的应用层的第 2 个特点是服务器集群中的 Session 管理。

业务总是有状态的，比如交易类网站需要有购物车记录用户购买的商品。在 Web 应用中，这些状态就被称为 Session。在单机情况下，Session 可以由部署在服务器上的 Web 容器（如 Tomcat）进行统一管理。但在使用负载均衡的集群环境中，要保证每次请求都能够获得正确的 Session 就是个很复杂的话题。在集群环境中，有如下手段可以进行 Session 管理。

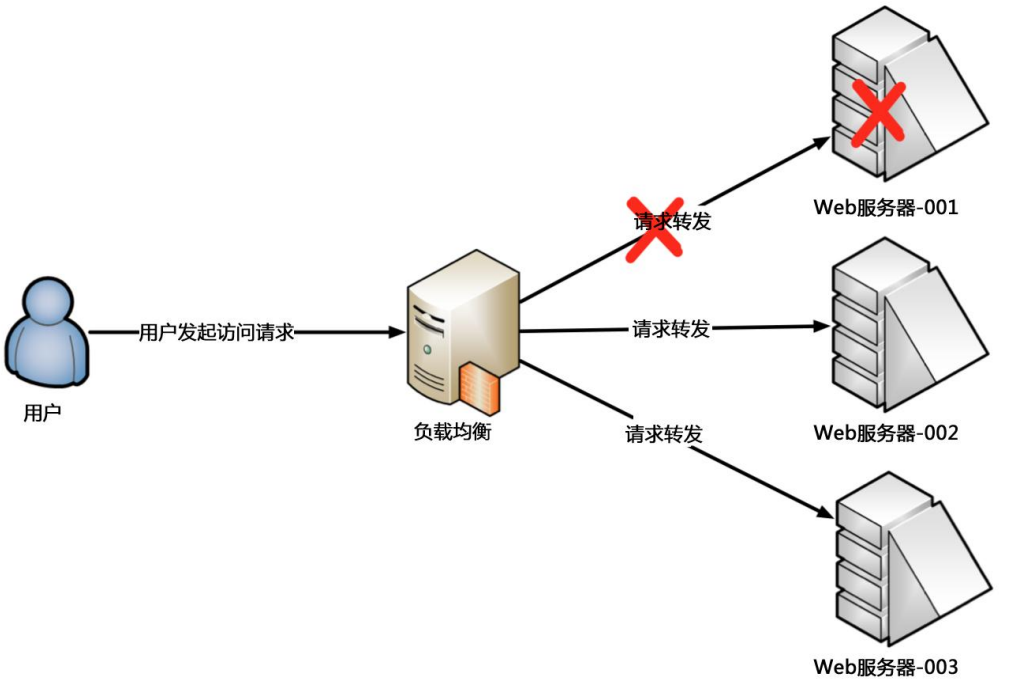


图 5.8

（1）复制 Session。应用服务器开启 Web 容器的 Session 复制功能，然后在集群中的几台服务器之间同步 Session 对象，使得每台服务器都保存所有用户的 Session 信息。

（2）绑定 Session。利用负载均衡的源地址 Hash 算法，将源于同一 IP 的请求分发到同一台服务器上。这样，在整个会话期间，用户的所有请求就都在同一台服务器上进行处理，即把 Session 绑定到某台特定的服务器上，这种方法又被称为“会话粘滞”。绑定 Session 的方案不符合系统高可用性的需求，因为一旦某台服务器宕机，该服务器上的 Session 就不存在了。即使将用户的请求切换到其他服务器，也会因为没有 Session 而导致业务无法正常处理。所以很少有网站使用绑定 Session 的方案。

（3）利用 Cookie 记录 Session。这种方式是将 Session 记录在客户端，在每次请求服务器时，都将 Session 放在请求中发送给服务器，在服务器处理后，再将修改过的 Session 发送回客户端。利用 Cookie 记录 Session 也有一些缺点，如下所述。

◎ Cookie 有大小限制，记录的信息有限。
◎ 每次请求响应都要传输 Cookie，影响性能。

◎ 如果用户关闭了 Cookie，访问就会变得不正常。

但 Cookie 简单易用，能够支持应用服务器的线性伸缩；而且大部分应用需要记录的 Session 信息又比较小。所以许多网站都或多或少地使用 Cookie 记录 Session。

（4）Session 服务器。利用独立部署的 Session 服务器或集群来统一管理 Session。应用服务器在每次读写 Session 时，都需要访问 Session 服务器。这样的方案是把应用服务器分为无状态的应用服务器和有状态的 Session 服务器。对于有状态的 Session 服务器，可以这样简单实现：在分布式缓存、数据库等组件的基础上进行封装，使其符合 Session 的存储与访问要求。

2. 高可用的服务层

可复用的服务模块为业务产品提供基础公共服务，通常都独立分布式部署，被具体应用远程调用。可复用的服务和应用一样，也是无状态的服务，因此可以使用类似负载均衡的失效转移策略实现高可用的服务。

除此之外，具体实践中，还有如下几种高可用的服务策略。

（1）分级管理。在运维层面对服务器进行分级管理，核心应用与服务优先使用更好的硬件。在服务部署上进行必要的隔离，避免故障的连锁反应。低优先级的服务通过启动不同的线程或部署在不同的虚拟机上进行隔离。高优先级的服务需要部署在不同的物理机上，核心服务和数据最好部署在不同地域的数据中心上。

（2）超时设置。在应用中设置调用服务的超时时间，一旦超时，通信框架就会抛出异常。应用根据服务调度策略，可以选择继续重试或者将请求转移到提供相同服务的其他服务器上。

（3）异步调用。应用对服务的调用，通过消息队列的异步方式实现。应用程序会将用户注册信息发送给消息队列服务器，之后立即返回用户注册成功的响应。而多个消费者任务（记录用户信息到数据库、发送注册成功邮件、开通权限）会从消息队列中获取用户注册的信息，之后异步执行。注意：不是所有的服务调用都可以使用异步调用模式，比如获取用户信息的服务或必须确认服务调用成功才能继续下一步操作的服务等，都不适合使用异步调用模式。

（4）服务降级。在网站访问的高峰期，服务可能因为大量的并发调用导致性能下降，严重时甚至会导致服务宕机。所以为了保证核心应用与服务的正常运行，我们必须对某些

服务进行降级。降级有如下两种手段。

◎ 拒绝服务：拒绝低优先级应用的调用，减少并发数，以确保核心应用；或随机拒绝部分请求，以节约资源。
◎ 关闭功能：关闭部分不重要的服务，或关闭服务内不重要的功能，以节约资源。

（5）幂等性设计。应用在调用服务失败后，会将请求发送给其他服务器，但这个失败可能不是真的！比如服务实际上已处理成功，但因为网络故障，应用没有收到响应，那么这时应用所重新提交的请求，就是在重复调用服务！如果这是一个转账服务，那么后果会很严重。重复调用服务是不可避免的，所以服务层必须保证重复调用服务与调用一次服务的结果是相同的，即服务具有幂等性。比如转账交易的操作就需要通过交易编号对调用的服务进行有效性校验，保证调用是有效的才能继续执行。

3. 高可用的数据层

对于系统而言，最珍贵的资产是多年运营积累下来的数据（用户数据、交易数据和商品数据等），所以保护了数据就是保护了企业的命脉。而保证数据高可用的手段是数据备份和失效转移，如下所述。

（1）数据备份：保证数据有多个副本，任意副本的失效都不会导致数据的永久丢失。

（2）失效转移：当有一个数据副本不可访问时，可以快速切换到其他副本。

注意：缓存服务不是数据存储服务，缓存服务器宕机引起的缓存数据丢失及服务器负载过高的问题，应该通过其他手段进行解决。可以让整个网站共享一个分布式缓存集群，应用只需要向共享缓存集群申请缓存资源即可，这样任何一台缓存服务器宕机所引起的缓存失效，都只会影响到缓存数据的一小部分，而不会对应用的性能和数据库负载造成太大的影响。

下面讲讲 CAP 的原理。为了保证数据的高可用，系统通常会牺牲数据的一致性。高可用的数据有如下几层含义。

◎ 数据持久性：保证数据可持久存储，即在将数据写入可持久性存储的硬件的同时，将数据备份成一个或多个副本。
◎ 数据可访问性：将多个数据副本存储在不同的存储设备上，如果一个存储设备损坏，就需要将对数据的访问切换到另一个数据存储设备上。这个过程要尽量快，让客户不可感知。

◎ 数据一致性：在数据有多个副本的情况下，如果网络、服务器或软件出现故障，则会导致部分副本写入成功，部分写入失败，这样就会造成各个副本之间存在数据不一致的情况。

下面讲讲备份数据。早期的数据备份主要是冷备份，即定期地将数据复制到存储介质中（磁带、光盘等）并进行物理存储，冷备份的优点是简单、廉价，缺点是不能保证数据最终一致。因为数据是定期复制的，所以备份设备中的数据比系统中的数据老，如果系统数据发生丢失，那么从上一个备份点开始，更新的数据就会永久丢失、不能恢复。

冷备份作为一个传统的数据保护手段，依然在网站日常运维中使用，但还需要进行数据热备份，以提供更好的数据可用性。数据热备份有两种：异步热备和同步热备。

下面讲讲失效转移。失效转移指的是在服务器集群中如果有任意一台服务器宕机，那么应用针对这台服务器的所有读写操作，都需要被重新路由到其他服务器，保证服务依然是可用的。保证数据高可用性的手段是失效确认、失效转移和数据恢复。

（1）失效确认。系统通过心跳检测和应用访问失败的报告来确认某台服务器是否宕机。对于应用程序的访问失败报告，控制中心还需要再发送一次心跳检测对服务器进行确认，以免应用程序判断错误。因为一旦进行失效转移，就会出现存储的多份数据副本不一致，后续就要对这种情况进行一系列复杂的纠错操作。

（2）访问转移。在确认服务器宕机后，就要将数据的读写访问重新路由到其他服务器上。这又分为如下两种情况。

◎ 存储完全对等：直接切换到对等服务器上。
◎ 存储不对等：重新计算路由，选择健康的服务器。

（3）数据恢复。在某台服务器宕机后，数据存储的副本就会减少，这时系统会从健康的服务器上复制数据，将副本的数量恢复到系统设定的值。

5.5.3 高可用质量保证

下面会介绍一些保证线上系统的高可用质量的手段。

（1）发布系统。系统的发布实际上与服务器的宕机效果差不多，因为需要关闭服务器上的原有应用，然后重新部署、启动新的应用。但系统的发布毕竟是一次提前预知的“服务器宕机”，其过程可以更柔和，对用户的影响可以更小。因为每次关闭的服务器只是集

群中的一小部分，而且在发布完成后可以立即访问，所以整个发布过程并不影响用户的使用。

（2）自动化测试。代码在发布到服务器之前需要进行严格的测试。即使发布的新功能都在原有系统的功能上进行小幅增加，也需要对整个网站的功能进行全面的回归测试。此外，还要针对各种浏览器进行兼容性测试。大部分系统都采用了 Web 自动化测试技术。

（3）预发布过程。即使经过了严格测试，在部署到现网后还是有可能出现问题的。因为测试环境与现网环境不同，特别是应用可能会依赖其他服务，例如数据库、缓存及第三方服务（短信网关、支付接口等），所以在发布时可以先将包发布到“预发布服务器”上，这样开发人员与测试人员就可以在预发布服务器上进行验证了。一般会测试一些典型的业务流程，在确认系统没有问题后才会正式发布。

（4）代码控制。代码控制的核心问题是如何进行代码管理，既能保证代码发布的版本稳定、正确，又能保证不同团队的开发互不影响。

（5）很多公司都要求周期性发布系统，例如要求每周四都是发布日，在月底、节日前不允许发布。这样做是为了避免周末、节假日出现问题时无人解决问题，也避免周末、节假日加班。使用“火车发布模型“可以有效控制发布故障，减少发布日的加班问题。

（6）灰度发布模型。灰度发布又叫作 AB 测试，常被用于观察用户体验或测试新功能，即在线上环境中同时存在新老版本的系统，通过将部分用户导流到新版本中并监控用户的操作，来得到用户体验报告，这样就可以比较出用户对新老系统的满意度，为运营人员提供最真实的可信度数据。

5.5.4　系统运行监控

我们常常说：“不允许没有监控的系统上线”。系统运行监控对于系统运维和架构设计优化至关重要，没有监控的系统，是不完整的系统。

（1）采集监控数据。用户行为日志指的是用户在浏览器上的所有操作及用户所在的操作环境（操作系统与浏览器的版本、IP 地址、页面访问路径、页面停留时间等）。大型系统的用户日志数据量惊人，数据存储与计算的压力都很大，许多网站都逐步开发了基于实时计算框架 Apache Storm 的日志统计与分析工具。

（2）监控管理，如下所述。

◎ 系统告警。如果监控的某项指标超过了阈值，就意味着系统可能要出现故障，这时就要对相关人员进行告警。在监控管理系统中可以配置告警阈值和值守人员的联络方式（通过邮件、即时通信工具、手机短信等方式），及时发出预警。
◎ 失效转移。监控系统在发现故障时应该主动通知应用，及时进行失效转移。
◎ 自动优雅降级。优雅降级指的是网站为了应付突发的访问高峰，主动关闭一部分功能，释放系统资源。自动优雅降级的监控系统是一种柔性架构的理想状态：监控系统会实时监控所有服务器，如果发现某部分应用负载过高，就会适当卸载低负载应用的一部分服务器，重新安装和启动高负载的应用，使应用负载总体均衡。如果所有应用的负载都很高，就会自动关闭一部分不重要的功能，保证核心功能的正常运行。

5.6 可伸缩设计

系统的伸缩性是架构中最重要的技术，即使用服务器的集群功能，通过不断向集群中添加服务器来增强整个集群的处理能力。“伸”即架构的规模和服务器的规模不断扩大。

5.6.1 架构的伸缩性设计

架构的伸缩性设计可以分成两类，一类是根据功能进行物理分离实现伸缩，另一类是单一功能通过集群实现伸缩。前者是不同的服务器部署不同的服务，提供不同的功能；后者是集群内的多台服务器部署相同的服务，提供相关的功能。

架构的物理分离分为如下两种。

◎ 纵向分离：根据功能模型的不同将系统分离，一般分为展示层、应用层、服务层和数据层。
◎ 横向分离：根据业务模型的不同，可以将业务纵向进行分离。

对集群规模的调整指通过负载均衡动态增减服务器的数量，如图 5.9 所示。

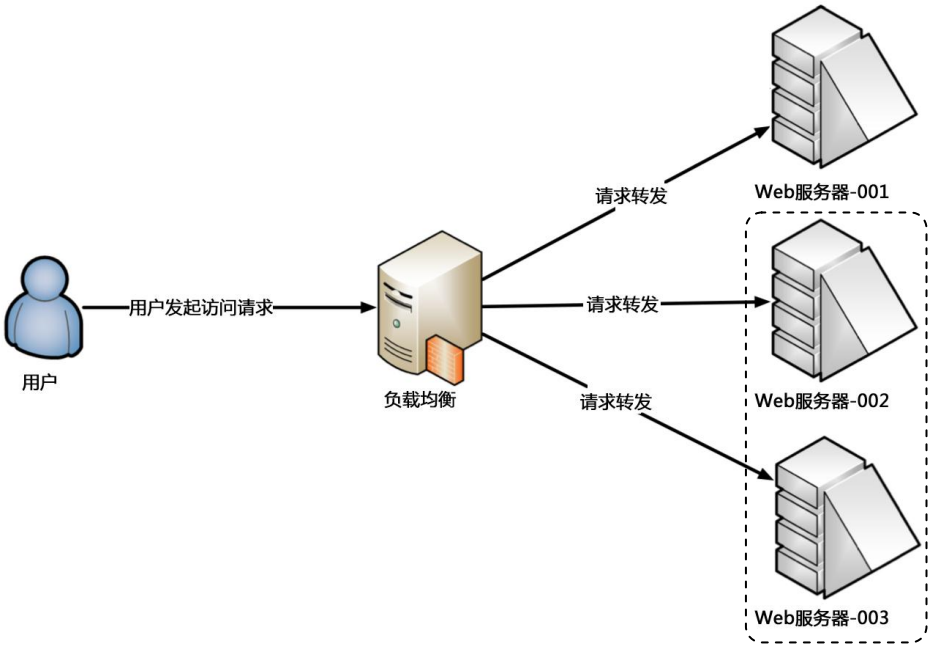


图 5.9

5.6.2　应用服务的伸缩性设计

应用服务器的伸缩性，指的是 HTTP 请求分发装置可以感知或者配置集群的服务器数量，还可以及时发现集群中新上线或下线的服务器，并能向新上线的服务器分发请求，停止向已下线的服务器分发请求。这种 HTTP 请求分发装置被称为负载均衡服务器。

负载均衡是网站必不可少的基础技术，不但可以实现网站的伸缩性，还可以改善网站的可用性，是网站的重要技术。负载均衡的具体技术实现也多种多样，从硬件实现到软件实现，从商业产品到开源，应有尽有，但实现负载均衡的基础技术不外乎如下几种。

（1）HTTP 重定向负载均衡。HTTP 重定向服务器是一台普通的应用服务器，其唯一功能就是根据用户的 HTTP 请求计算出一台真实的服务器地址，并将该服务器地址写入 HTTP 重定向响应中（重定向响应状态码为 302）返回给用户的浏览器。用户的浏览器在收到响应之后，根据返回的信息重新发送一个请求到真实的服务器上。

（2）DNS 域名解析负载均衡。在利用 DNS 处理域名解析请求的同时进行负载均衡是

另一种常用的方案。每次域名解析请求都会根据负载均衡算法计算一个不同的 IP 地址并返回，这样在 A 记录中配置的多台服务器就构成了一个集群，可以实现负载均衡。DNS 域名解析负载均衡的优点是将负载均衡工作交给 DNS，省略了网络管理的麻烦，缺点就是 DNS 可能缓存 A 记录，不受网站控制。事实上，大型网站总是部分使用 DNS 域名解析，作为第 1 级负载均衡手段，再在内部做第 2 级负载均衡。

（3）反向代理负载均衡。反向代理可以缓存资源，改善网站性能。事实上，反向代理也可以做负载均衡，由于反向代理服务器在 HTTP 层面转发请求，因此也叫作应用层负载均衡，其优点是部署简单，缺点是可能导致成功系统的瓶颈。

（4）IP 负载均衡。指网络层通过修改请求目标地址进行负载均衡，在用户请求数据包到达负载均衡服务器后，负载均衡服务器在操作系统内核获取网络数据包，根据负载均衡算法计算得到一台真实的 Web 服务器的地址，然后将数据包的 IP 地址修改为真实的 Web 服务器的地址，不需要通过用户进程处理。在真实的 Web 服务器处理完毕后，相应的数据包回到负载均衡服务器，负载均衡服务器再将数据包源地址修改为自身的 IP 地址发送给用户的浏览器。

这里的关键在于真实的 Web 服务器的相应数据包如何返回给负载均衡服务器：一种方案是负载均衡服务器在修改目标 IP 地址的同时修改源地址，将数据包源地址改为自身的 IP，即源地址转换（SNAT）；另一种方案是将负载均衡服务器同时作为真实物理服务器的网关服务器，这样所有的数据都会到达负载均衡服务器。

IP 负载均衡在内核进程中完成数据分发，较反向代理均衡有更好的处理性能。但由于所有请求响应的数据包都需要经过负载均衡服务器，因此负载均衡的网卡带宽成为系统的瓶颈。

（5）数据链路层负载均衡。这种数据传输方式又被称为三角传输模式，负载均衡数据在分发过程中不修改 IP 地址，只修改目标 MAC 地址，通过将真实物理服务器集群所有机器的虚拟 IP 地址和负载均衡服务器的 IP 地址配置为一致的 IP 地址，来达到负载均衡，这种负载均衡方式又被称为直接路由方式。在用户的请求到达负载均衡服务器后，负载均衡服务器将请求数据的目标 MAC 地址修改为真实的 Web 服务器的 MAC 地址，不修改数据包目标 IP 地址，因此数据可以正常到达目标 Web 服务器，该服务器在处理完数据后可以经过网管服务器而不是负载均衡服务器直接到达用户的浏览器。

5.6.3　分布式缓存的伸缩性设计

在分布式缓存服务器集群中，不同服务器中的缓存数据并不相同，对缓存的访问请求，不能在缓存服务器集群中的任意一台上处理，必须先找到缓存中有需要数据的服务器，才能进行访问。这会严重制约分布式缓存集群的伸缩性设计，因为新上线的缓存服务器没有缓存数据，而已下线的缓存服务器还缓存着网站的许多热点数据。

分布式缓存集群伸缩性设计的最主要目标就是让新上线的缓存服务器对整个分布式缓存集群的影响最小，也就是说在新加入缓存服务器后，应使在整个缓存服务器集群中已经缓存的数据尽可能被访问到。

5.6.4　数据存储的伸缩性设计

数据存储服务器集群的伸缩性对数据的持久性和可用性都提出了更高的要求。具体来说，可以分为关系数据库集群的伸缩性设计和 NoSQL 数据库的伸缩性设计。

在关系数据库集群的伸缩性设计中，主要的关系数据库都支持数据复制功能，使用这个功能可以对数据库进行简单伸缩。另外，除了利用数据库进行主从读写分离，也可以利用业务分隔模式使不同业务的数据表部署在不同的数据库集群上，即数据分库。但这种方式的制约条件是跨库不能进行 Join 操作。在大型网站的实际应用中，即使使用了分库和主从复制，对一些单表数据量很大的表还需要进行分片，将一张表拆开且分别存储在多个数据库中。

NoSQL 主要指非分布式的非关系数据库设计模式。一般而言，NoSQL 数据库产品都放弃了关系数据库的两大重要基础：以关系代数为基础的结构化查询语言（SQL）和事物一致性保证（ACID），但是强化了高可用性和可伸缩性。

5.7　可扩展性设计

扩展性是指在对现有系统影响最小的情况下，系统功能可持续扩展或提升的能力。设计架构可扩展的核心思想是模块化，并在此基础上降低模块间的耦合性，提供模块的复用性。对模块通过分布式部署，将独立的模块部署在独立的服务器上（集群），从物理上分离模块之间的耦合关系。模块在分布式部署后的具体聚合方式主要有分布式消息和分布式服务这两种。

5.7.1 通过消息降低系统的耦合性

如果在模块之间不存在直接调用，那么新增模块或者修改模块对其他模块影响最小，这样系统的可扩展性无疑更好一些。事件驱动框架指通过在低耦合的模块之间传输事件消息，以保持模块的松散耦合，并借助事件消息的通信完成模块间的合作，典型的架构就是生产者消费者模式。在大型网站架构中具体的实现手段很多，最常用的就是分布式消息队列，消息队列利用发布–订阅模式工作：消息发送者发布消息，一个或者多个消息接收者订阅消息。由于消息发送者不需要等待消息接收者处理数据就可以返回，所以系统具有更好的响应延迟；同时，在网站访问高峰期间，消息可以被暂时存储在消息队列中等待处理，减轻了数据库等后端存储的负载压力。

5.7.2 通过分布式服务提供可复用的业务

使用分布式服务是降低系统耦合性的另一个重要手段。例如，分布式消息队列通过消息规避系统的耦合性，由不同的子系统处理同一个消息；分布式服务则通过接口分解系统的耦合性，不同的子系统通过相同的接口描述进行服务调用。大型系统的分布式服务的需求与特点为：负载均衡、失效转移、高效的远程通信、整合异构系统、对应用最小入侵、版本管理和实时监控。

目前，国内有较多成功实施案例的开源分布式服务框架是阿里巴巴的 Dubbo，服务消费程序通过服务接口使用服务，而服务接口通过代理加载具体服务，具体服务可以是本地的代码模块，也可以是远程的服务，因此对应用入侵较小；应用程序需要调用服务接口，服务框架则根据配置自动调用本地或远程实现。

服务框架的客户端模块通过服务注册中心加载服务提供者列表（服务提供者在启动后主动向服务注册中心注册自己可提供的服务接口列表），查找需要的服务接口，并根据配置的负载均衡策略将服务调用请求发送到某台服务提供者的服务器。如果服务调用失败，则客户端模块会自动从服务提供者列表中选择一个可提供同样服务的另一台服务器重新请求服务，实现服务的自动失效转移，保证高可用服务。

5.7.3 建立开放平台

大型系统为了更好地服务自己的用户，开放了更多的增值服务，会将系统内部的服务封装成一些调用接口开放出去，供外部的第三方开发者使用，这种提供开放接口的平台被

称作开放平台。开放平台是系统内部和外部交互的接口，在外部需要面对众多的第三方开发者，在内部需要面对系统内诸多的业务服务。虽然每个系统的业务场景和需求都不相同，但开发平台的架构设计大同小异。开发平台的要求如下。

◎ API 接口：是开发平台暴露给开发者使用的一组 API，其形式可以是 RESTful、WebService 和 RPC 等形式。
◎ 协议转换：将各种 API 输入转换成内部服务可以识别的形式，并将内部服务的返回封装成 API 格式。
◎ 安全：除了一般应用需要的身份识别、权限控制等安全手段，开放平台还需要配置分级的访问带宽限制，以保证资源被公平合理地使用。
◎ 审计：记录第三方应用的访问情况并进行监控、计费等。
◎ 路由：将开放平台的各种访问路由映射到具体的内部服务。
◎ 流程：将一组离散的服务组织成一个上下文相关的新服务，隐藏服务的细节，提供统一的接口供开发者调用。

5.8　架构设计的典型案例——订单系统

如图 5.10 所示，订单系统作为电商系统的“中轴线”，贯穿了整个电商系统的全部流程，电商系统中的核心系统都是围绕订单构建的，订单系统也是随着电商行业的发展逐渐演变的。

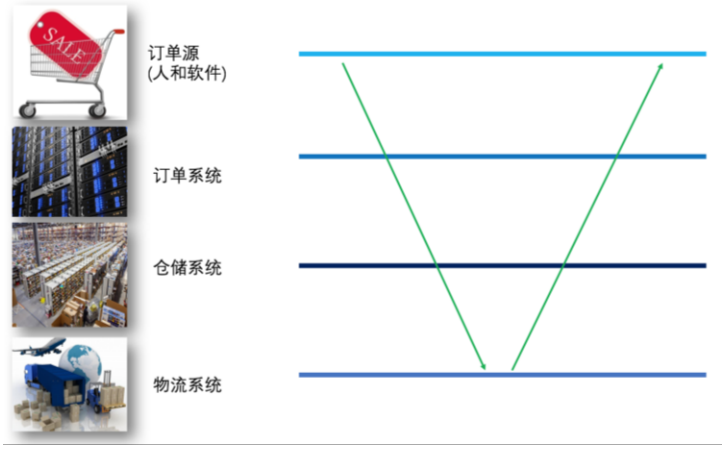


图 5.10

订单系统用于管理订单类型和订单状态，收集商品、优惠、用户、收货信息、支付信息等一系列订单实时数据，并进行库存更新、订单下发等一系列动作。订单系统业务的基本模型涉及用户、商品（库存）、订单、付款，订单基本流程是下订单→减库存，下订单和减库存必须同时完成，否则，若下了订单但是没减库存，就会导致超卖，使商家库存不足，消费者下了订单却买不到东西，体验不佳；若减了库存但是没有生成订单，就会导致少卖，使商家库存积压或者反复修改商品信息，体验也不佳。

5.8.1 订单的基本概念

在设计订单系统时需要考虑订单字段、订单信息、用户信息、商品信息、金额信息、时间信息等内容，这些内容决定了订单系统的稳定性和可持续性。因为订单的来源和操作的多样性等，订单有多样性的特点，如图 5.11 所示。

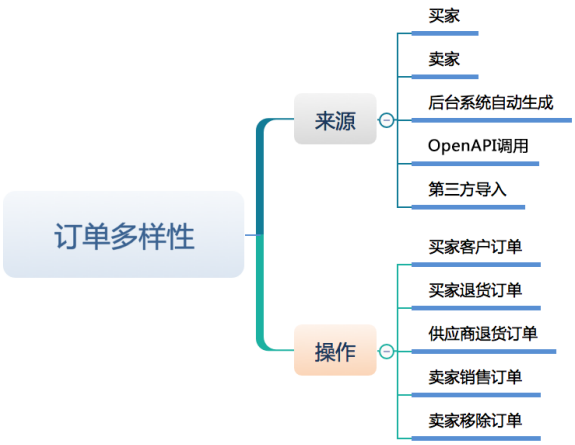


图 5.11

1. 订单字段

订单字段包含需要在订单中记录的信息，主要用于其他系统如商品系统、物流系统、支付系统等的沟通，为下游系统提供信息依据，如图 5.12 所示。

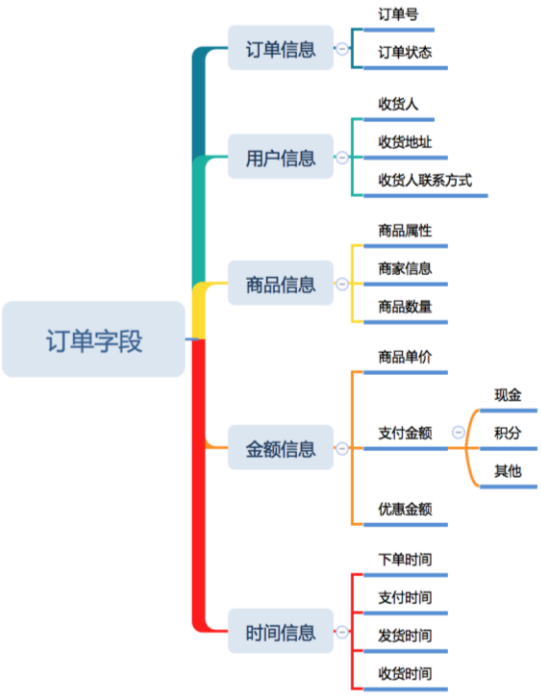


图 5.12

2. 订单记录信息

订单号作为订单识别的标识，往往由一串数字组成，根据订单的增加进行自增，也可以在设计订单号时考虑订单的加密设置（否则其他人通过订单编号就能计算销售量）。订单号后续用作订单的唯一标识，用于 WMS（仓存管理系统）和 TMS（运输管理系统）时的订单识别。

3. 用户信息

用户信息指购买人的相关信息，主要包括姓名、地址和手机号。在 WMS（仓储管理系统）和 TMS（运输管理系统）中，地址信息会被用于区分区域和配送安排。

4. 商品信息

商品信息一般包括商品的基本信息和库存，主要影响库存更新和 WMS 生产。“金额”在逻辑上其实也属于商品信息的范畴，但由于其比较特殊，所以本节将其单独拿出来讲解。

其中，金额信息用于记录订单在每个状态节点的触发时间。在订单产生的商品信息里除了要记录最终金额，还要记录过程金额，比如商品分摊的优惠金额、支付金额、应付金额等。在后续的订单结算、退换货、财务等环节中都要用到。时间信息用于记录订单每个状态节点的触发时间如创建时间、支付时间和退款时间等。

5.8.2 订单流程

订单流程指整个订单从产生到完成的整个流程，包括正向流程和逆向流程。

1. 正向流程

正向流程如图 5.13 所示。

订单流程涉及的环节非常多，有如下细节需要注意。

◎ 释放库存的场景：超时未支付、订单自动取消。
◎ 如果选择 COD（货到付款），则支付环节相应地被转移到订单配送之后，在该过程中所有与款项相关的逻辑都变为只操作金额数字，不对结算和账户进行打款和退款操作。
◎ 金额需要分摊到子商品订单，方便后期订单的退款退货。
◎ 订单系统审核的一个作用是对恶意用户或者刷单情况进行处理。系统可根据白名单、黑名单、消费频次、促销品购买量等设计风控规则。如果后续会进入人工审核环节，则可以在规则上适当放宽。当触发规则需要进行订单退订操作时，在设计上要考虑对用户体验的影响，在前台文案上说明当前节点为审核状态或者等待接单状态。
◎ 在 O2O 领域有催单的概念，传统电商则通过关联第三方物流信息进行跟踪。催单触发一般会设定一定的时间间隔，在特定的时间内只触发一次催单请求。
◎ 需要将预售等货和移仓做成 SOA 服务，以便在交易页面计算预计时间和预计到货时间。移仓处理依赖仓库的情况，也会涉及后续拆分和合并包裹的逻辑。
◎ 在生产订单时要先判断报缺情况，如果出现报缺问题，则要考虑整单报缺、部分报缺、换货或者换转退的情况（库存、仓促调拨和退款）。报缺情况分为系统报缺和实物报缺，这是相互承接但相对独立的两个环节。
◎ 电商系统要考虑 7 天无理由退货的场景，即在订单状态完成后申请退货。此时主要涉及的是金额上的计算及一些财务程序（如发票等）问题的处理。

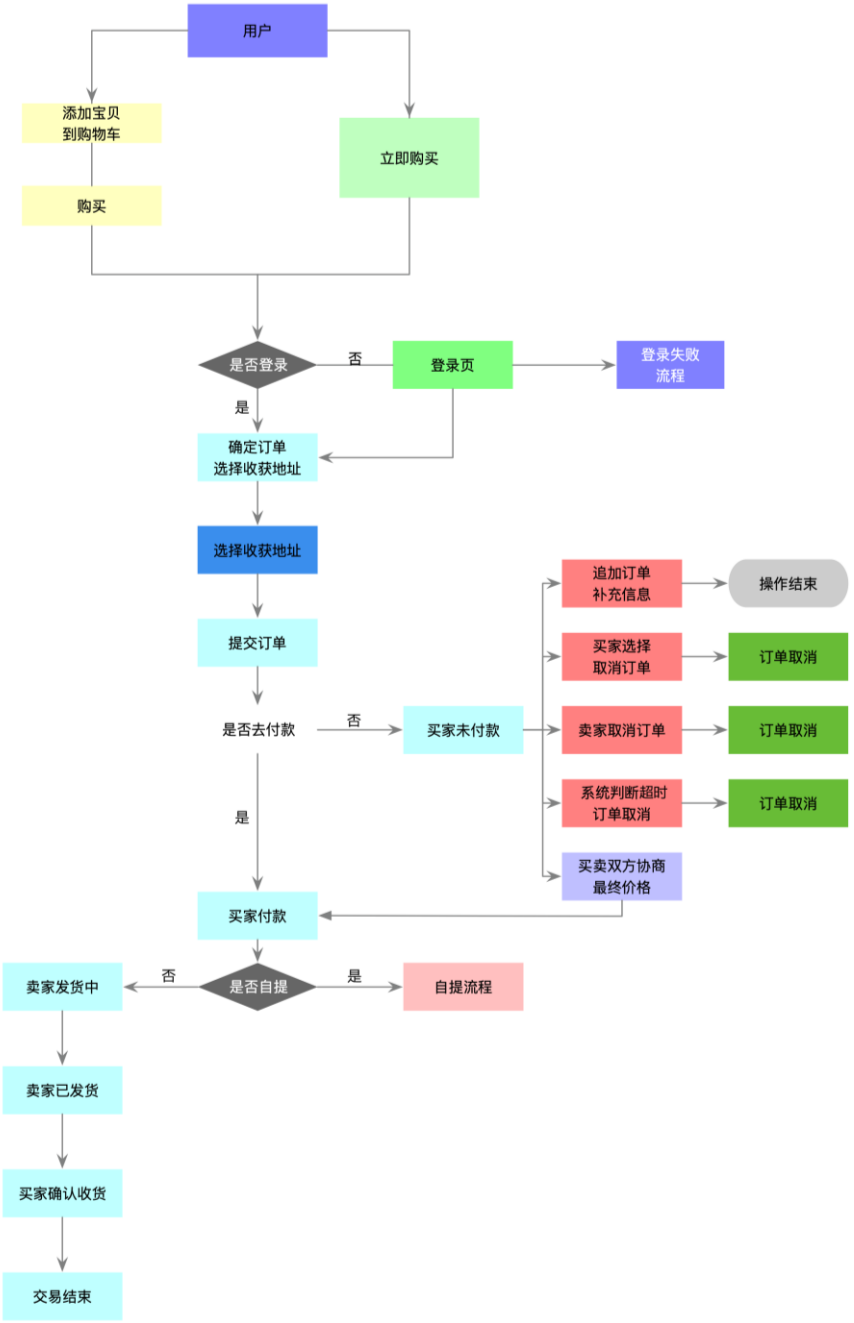

用户
添加宝贝
到购物车
购买
立即购买
是否登录
否
登录页
登录失败
流程
是
确定订单
选择收获地址
选择收获地址
提交订单
是否去付款
否
买家未付款
追加订单
补充信息
操作结束
买家选择
取消订单
订单取消
卖家取消订单
订单取消
系统判断超时
订单取消
订单取消
买卖双方协商
最终价格
是
买家付款
卖家发货中
否
是否自提
是
自提流程
卖家已发货
买家确认收货
交易结束

图 5.13

2. 逆向流程

逆向流程指因订单取消、退货及退款等情况引发的整个流程。逆向流程的触发主要有如下几种情况，如图 5.14 所示。

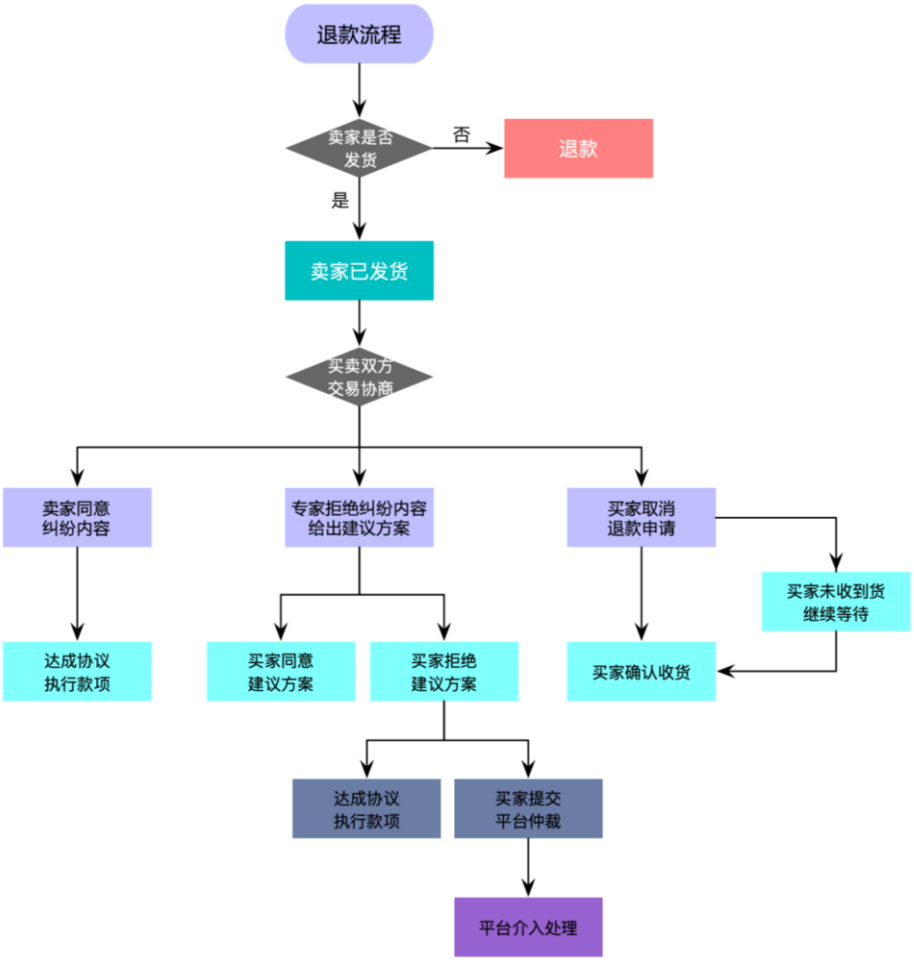


图 5.14

（1）用户自主取消订单（整单）。

（2）风控系统触发取消订单（整单）。

（3）在客服接到客户投诉仲裁后触发取消订单（整单）。

（4）超时未支付取消订单（整单）。

（5）换货报缺转为退单（整单、部分报缺）。

逆向流程触发的关注点如下。

（1）订单状态（某一节点如订单产生后不允许取消订单）。

（2）当退单被商家拒绝后需要转入客服仲裁环节。

（3）部分退款依然享有优惠活动，金额按照分摊进行退款。

3. 订单状态

下面从订单状态的设计目的和存在价值去分析和理解它的设计机制，比如维度及维度的颗粒度大小。

正向流程和逆向流程的维度（各选取主要状态）如下。

◎ 正向订单：已锁定、已确认、已付款、已发货、已结算、已完成、已取消等。
◎ 正向预售订单：预付款已付未确认、已确认未付尾款（变更）。
◎ 正向问题单：未确认、未锁定、未发货、部分付款、未付款等。
◎ 逆向退单：待结算、未收到货、未入库、质检不通过、部分收货、已取消、客户已收货等。
◎ 逆向换单：完成、已结算、客服已收货等。

服务对象的维度如下。

◎ 顾客/用户：待付款、待发货、待收货、待评价、买家已付款、交易成功或失败、卖家已发货、退款成功、交易关闭等。
◎ ERP 等其他交互系统：已锁定、已确认、已分仓、已分配、已出库、已收货、已完成等。
◎ 客服：等待买家付款、待付款和待发货的订单、退款中的订单、定金已付、买家已付款、卖家已发货、交易成功、交易失败、异常订单。

4. 订单推送

当状态机发生变化时，需要将对应的变化告知相关人员以便了解当前订单的情况，这就是订单推送的作用。

订单推送的触发依赖于状态机的变化，涉及的信息有：

◎ 推送对象（用户、物流人员、商家）；
◎ 推送方式（App 端的消息、微信端的消息、短信）；
◎ 推送节点（状态改变）。

5.8.3 订单系统设计中的挑战和实践

下面从订单系统需求的演变和订单系统架构的演变这两方面讲讲订单系统设计中的挑战和实践。

1. 订单系统需求的演变

订单系统需求的演变一般会经历 3 个阶段，如下所述。

第 1 阶段，实现购买流程。

◎ 实现订单的创建、发货、确认等信息闭环。
◎ 支持订单审核（在初期支持人工审核即可）。
◎ 支持在用户端显示订单相关信息。
◎ 支持促销金额的计算。

第 2 阶段，提供服务。

◎ 提供订单分布式服务。
◎ 支持跨平台交易单的生成（即在同一个大交易单内既有商家商品又有自营商品或者多个商家的商品）。
◎ 支持拆单、合单逻辑（配送单、支付单等）。
◎ 提供多样的订单推送服务来完善订单的状态。

第 3 阶段，支持不同营销手段下的订单类型。当平台发展到足够大的规模时，提效、稳定就变成了一个重要的话题，平台应该可以提供不同营销场景下的订单如团购、预购、秒杀等。

2. 订单系统架构的演变

随着业务需求、用户量及各种系统等因素的改变，订单系统的架构也要随之改变。下

面讲解订单系统的三代演变及最佳实践。

第 1 代订单系统的架构：简单粗暴

如图 5.15 所示，第 1 代订单系统的架构简单粗暴。

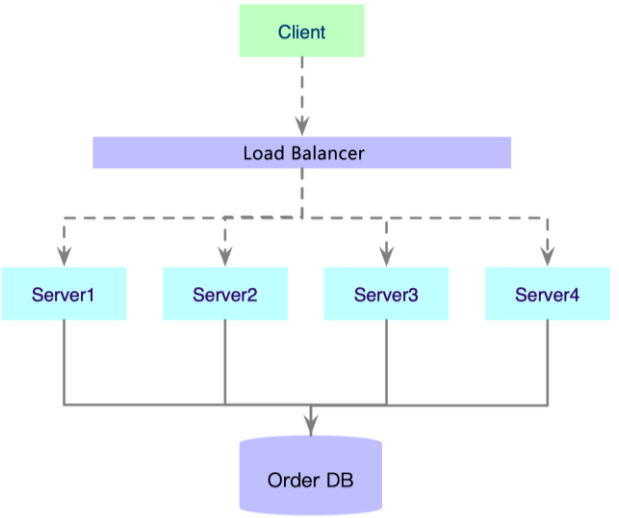


图 5.15

在第 1 代订单系统的架构中，订单状态是在特定的服务器上进行处理的，服务一旦出现问题，就会造成订单的丢失，导致订单流无法进行下去。

第 1 代订单系统的缺点如下。

◎ 服务单点。
◎ 数据库单点。

第 2 代订单系统的架构：无状态异步驱动

如图 5.16 所示，第 2 代订单系统的架构为无状态异步驱动。

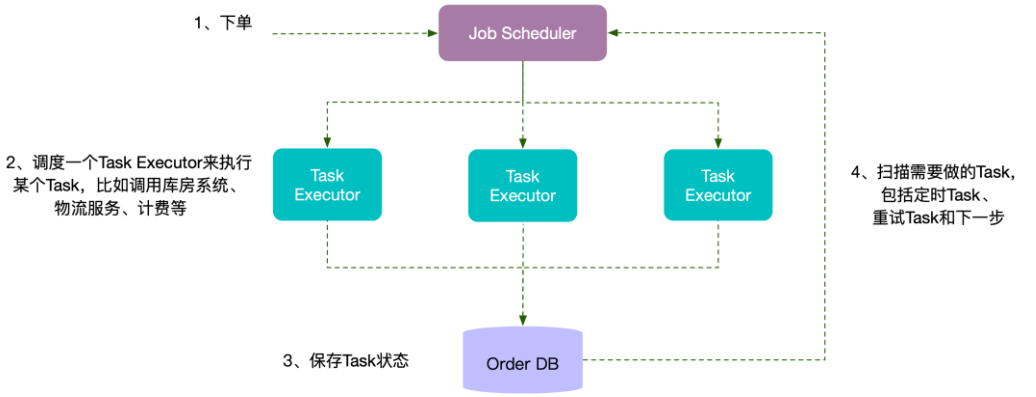


图 5.16

在第 2 代订单系统的架构中，应用服务器不再保留订单状态，但给数据库服务器造成了高频查询压力，数据库相对比较脆弱。

第 2 代订单系统二代的缺点为状态扫描带来的服务负载。

第 3 代订单系统的架构：队列模式

如图 5.17 所示，第 3 代订单系统的架构为队列模式。

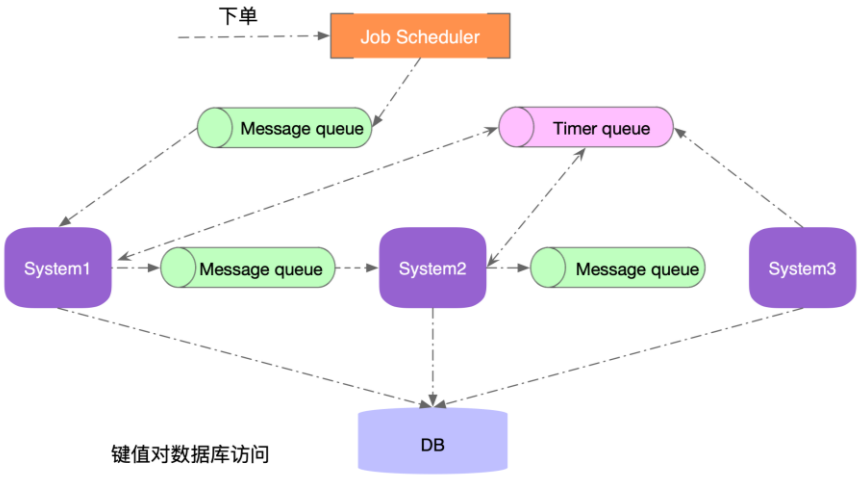


图 5.17

在第 3 代订单系统的架构中，订单的状态的改变不再依靠高频查询数据库来获得，而是通过队列模式来获得，这很好地减轻了数据库的压力。但是在该系统中，System2 成了核心，该模块的维护会变得很复杂，这也是架构设计的关键：如果没有完美的架构，则只能得到一个平衡的架构。

3. 系统优化的重要实践

实践 1，重试和补偿。

◎ 多个机器重试不能同步，需要随机跳跃（Jitter）和指数回退（Exponential Back-off）。
◎ 正在重试的服务也可能宕机，需要保存状态（State）。

实践 2，幂等性。

◎ 没收到响应不见得失败了。
◎ 自己响应了不见得别人以为自己成功了。
◎ 重试时必须带上唯一的有意义的 ID。
◎ 每一个服务的调用都必须是幂等的。
◎ 非只读的服务必须保存状态。

实践 3，一致性实践。

◎ 订单系统有强一致性需求。
◎ 无单点故障的分布式系统的一致性是难以解决的问题。
◎ 已有 Paxos 算法，现有开源系统（例如 ZooKeeper）。
◎ 可以使用常用的、成熟的关系数据库来避免单点故障的发生。
◎ 云端分布式无单点故障的系统。

实践 4，工作流（Workflow）。

◎ 可扩展性：无状态的服务、分布式部署、分布式存储、工作流状态。
◎ 可靠性：定时器、重试、幂等性、强一致性的状态。
◎ 可维护性：工作流的描述与执行 Activity 的描述相分离，支持异步触发。
◎ 支持版本和升级。

5.8.4 数据库优化

下面从数据库读写分离和数据库分库分表这两方面讲讲如何进行数据库优化。

1. 数据库读写分离

数据库读写分离的基本原理是让主数据库处理事务性查询，让从数据库处理 Select 查询，当然主服务器也可以提供查询服务。读写分离的最大作用是降低服务器的压力。

数据库读写分离的好处如下。

◎ 增加了冗余。
◎ 增加了机器的处理能力。
◎ 对于以读操作为主的应用，使用读写分离是最好的场景，因为可以确保写的服务器压力更小，而读又可以接受时间上的延迟。

提高读写分离性能的主要原因如下。

◎ 物理服务器增加、负荷增加。
◎ 主数据库和从数据库只负责各自的写和读，可很大程度地缓解 X 锁和 S 锁争用。
◎ 从数据库可配置 MyISAM 引擎，提升查询性能及节约系统开销。
◎ 从数据库同步主数据库的数据与主数据库直接写还是有区别的，从数据库通过主数据库发送来的 binlog 恢复数据，但最重要的区别在于主数据库向从数据库发送 binlog 是异步的，从数据库恢复数据也是异步的。
◎ 读写分离适用于读多写少的场景，如果只有一台服务器，则当 Select 很多时，Update 和 Delete 会被这些 Select 访问中的数据堵塞，等待 Select 结束，并发性能不高。对于写和读比例相近的应用，应该部署双主相互复制。
◎ 可以在从数据库启动时增加一些参数来提高其读的性能，例如--skip-innodb、--skip-bdb、--low-priority-updates 及--delay-key-write=ALL。当然，这些设置也是需要根据具体的业务需求来做的，不一定能用上。
◎ 分摊读取。假如我们有 1 个主数据库和 3 个从数据库，假设现在 1 分钟内有 10 条写入和 150 条读取，那么 1 个主数据库和 3 个从数据库相当于共计 40 条写入，而读取总数没变，因此平均下来每台服务器都承担了 10 条写入和 50 条读取（主数据库不承担读取操作）。因此，虽然写入请求数没变，但是读取请求数大大分摊了，提高了系统的性能。另外，在读取请求被分摊后，又间接提高了写入的性能。所以，总体性能提高了，说白了就是拿机器和带宽换性能。

◎ MySQL 复制的另一大功能是增加冗余和提高可用性，在一台数据库服务器宕机后，能通过调整另一台从数据库服务器来以最快的速度恢复服务，因此不能光看性能，也就是说有 1 个主数据库和 1 个从数据库也是可以保证数据库稳定的。

实现方案如图 5.18 所示。

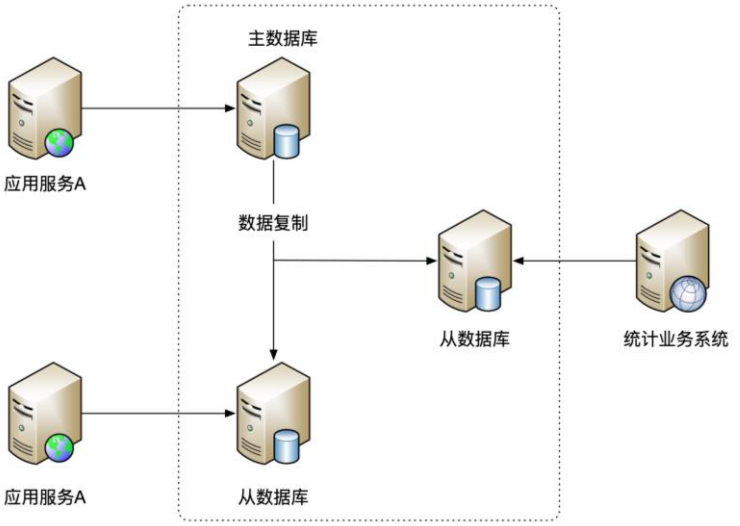


图 5.18

2. 数据库分库分表

不管采用哪种分库分表框架或者平台，其核心思路都是将原本保存在单表中的过大数据进行拆分，将这些数据分散保存到多个数据库的多个表中，避免因为单表数据量太大给数据的访问带来读写性能的问题。所以，在分库分表场景下最重要的一个原则就是将数据尽可能平均拆分到后端的数据库中，如果拆分得不均匀，则还会产生数据访问热点，同样存在热点数据因为增长过快使数据库单表数据量过大的问题。

而对于数据以什么样的纬度进行拆分，在很多场景下都是对业务数据的 ID（在大部分场景下，此 ID 都是自增长的方式）做 Hash 取模（哈希取模）的方式进行平均拆分的，这确实在很多场景下都是非常适用的，但并不是在所有场景下都是最优的选择。也就是说，如何拆分数据并没有所谓的金科玉律，更多地需要结合业务数据的结构和业务场景来决定。

以电商订单数据拆分为例，订单是任何一个电商平台都有的业务数据，每个平台的用户在提交订单时都会在平台后端生成订单相关的数据，一般记录一条订单数据的数据库表结构如图 5.19 所示。

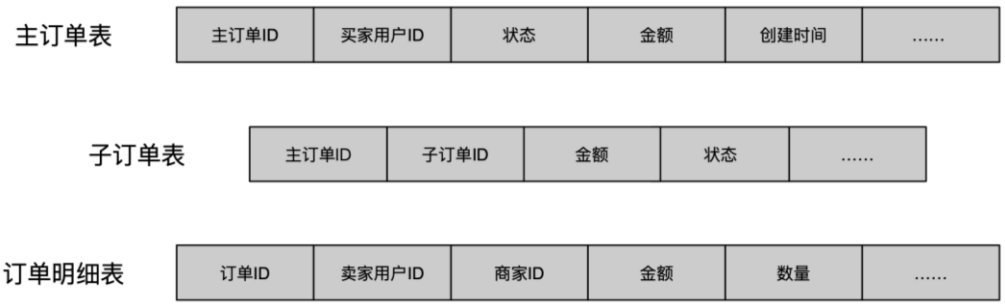


图 5.19

订单数据主要由三张数据库表组成，主订单表对应的就是用户的一个订单，每提交一次订单，都会生成主订单表的一条数据。在某些情况下，用户可能在一个订单中选择不同卖家的商品，每个卖家又会按照自己在该订单中提供的商品计算相关的商品优惠（如满 100 元减 10 元），以及按照不同的收货地址设置不同的物流配送，所以会出现子订单的相关概念，即一个主订单会由多个子订单组成，则真正对应到每个商品的具体订单信息被保存在订单详情表中。

如果一个电商平台的业务健康发展，则订单数据容易因为单个数据库表中的数据量过大而造成性能瓶颈，所以需要对其进行数据库拆分。此时从理论上对订单进行拆分是可以从两方面进行的，一方面是通过订单 ID（一般为自增长 ID）进行取模，即以订单 ID 为分库分表键，另一方面是通过买家的用户 ID 进行哈希取模，即以买家用户 ID 为分库分表键。

3. 两种方案的对比

如果通过订单 ID 进行取模，比如按 1024 进行取模，则可以保证主订单及相关子订单的订单详情数据平均落入后端 1024 个数据库表中，这在原则上很好地满足了数据尽可能平均拆分的原则。

如果通过买家的用户 ID 进行哈希后取模，比如也按照 1024 进行取模，则在技术上也能保证将订单数据拆分到后端的 1024 个数据库表中，通过用户 ID 进行哈希后取模会出现问题，也就是说如果某些卖家的交易量非常大，那么这些卖家的订单数据量（特别是订单

详情表的数据量）会比其他卖家多出不少，也就是说会出现数据不平均的现象，最终导致这些卖家的订单数据所在的数据库相对其他数据库提前进入数据归档。

为了避免在线交易数据库的数据量增大导致数据库的性能出现问题，我们一般将 3 个月内的订单数据保存在在线交易数据库中，将超过 3 个月的订单归档到后端的专门归档数据库中。

所以从对数据尽可能平均拆分这条原则来看，通过订单 ID 进行取模看起来更能保证订单数据的平均拆分，但我们暂时不要这么快下结论，要根据不同的业务场景和最佳实践多思考不同维度的优缺点。

5.8.5　小结

电商平台的需求一直在变化，订单系统的架构也会随之变化，架构设计就是一个持续改进的过程。如果想将订单系统做得更好，就需要更加深入系统的每一个环节，比如对容灾、灾备、分流、流控等都需要慢慢雕琢。在架构中没有完美的架构，只有平衡的架构，不要追求单点的完美，要追求多点的平衡。

第 6 章

架构的保障：质量与风险

6.1 内建质量体系

成本理论告诉我们，设计和开发是成本控制的源头，一般而言，成本随着项目的推进，在不同的阶段会呈现几何级的增长。实践证明，如果在前期设计时没有考虑周全，则用户往往抱怨不断，甚至导致项目返工。所以如何实现前期预防，对项目的成功至关重要，我们将这种前期预防的质量管理模式称为“内建质量（Built-in Quality）”。

内建质量是所有质量思维中最难实现的，也是质量挑战的终极目标。举个例子，一个简单的排线连接器是有正反两面的，如果将其正反面装反，则需要到电流测试时才能发现该问题，但如果将接头做成不对称的形状，在装反时让插头插不进去，就可以预防该问题的发生。这种防呆装置在组装设计上使用普遍，其实就属于内建质量的范畴。

每个公司都希望以最快的速度可持续地交付新功能，并应对瞬息万变的商业环境，这些都有赖于解决方案的质量。内建质量则有助于减少需求召回、返工及缺陷修复相关的延迟成本，非常适用于大规模系统，并且是强制性实施的。

内建质量强调“三不原则”：不接受不良品，不设计或制造不良品，不流出不良品，如图 6.1 所示。其特定范围可大可小，小到一个人，大到一个工厂。在流水线生产模式下，要想做到每个工位的内建质量是非常困难的，所以其下游布置了许多检测来找出这些不良品。从整个工厂来看，如果这些不良品都能被拦下来，不流到客户端，则也算是工厂整体做到了内建质量。虽然搭建一套真正持续有效的内建质量体系是极其困难的，但即便是一个很难企及的目标，我们也不该放弃。

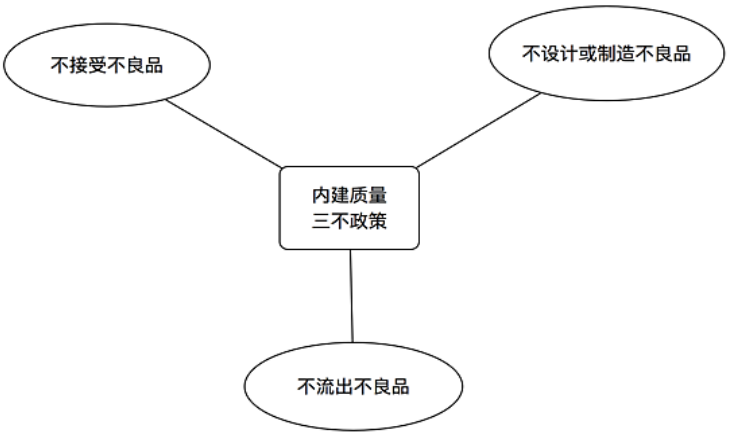


图 6.1

内建质量思维就是激发全员追求完美的工匠精神，唯有所有环节的人（包括企业家、领导）都能找回这种带有某种偏执狂的执着、认真、负责、追求完美的精神，才能将内建质量思维落实到业务当中，将其转化为企业的核心价值观。内建质量思维只有体现在产品和服务上，能让客户感受到，才能成为竞争对手极难模仿和跨越的核心竞争力。

“早发现、早解决”的难点在于发现而不是解决，质量管理的基本思路是上游管控，但要建立有效的“早发现、早解决”机制并不那么容易。就拿生产来说，对来料的抽检不能保证 100%上线的料没有不良品；生产线上的工人也不可能不出错；机器用久了更不会没有瑕疵；测试仪器也有校准或量测不到位的时候；品检也有眼花误判的时候，等等，这些都会导致不良品的流出。要想杜绝不良品的流出，就得从头到尾设置许多风险预警、识别装置等。

“早发现、早解决”体现在能够在表面正常的情况下嗅到发生危机的风险，从而及早采取防范或应变措施，大大降低发生损害的程度。所以，从事质量工作的人除了要能解决问题，还要能识别发生质量风险的征兆。

虽然质量改进永远不能停止，但是追求持续改进不能操之过急，应从小处着手，慢慢扩展。PDCA 循环正是推动持续改进的重要方法之一，如图 6.2 所示。

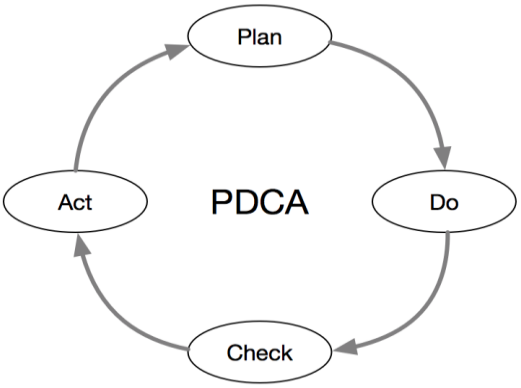


图 6.2

PDCA 代表了持续改进的如下 4 个主要步骤。

（1）P（Plan）：指仔细分析现状，并将现状与目标进行对比，看看缺口在哪里，然后详细规划改善的步骤。可以分几个步骤：一是将问题和目标定义清楚；二是利用各种工具，例如作业流程图、鱼骨图等，将问题相关的流程和因素梳理出来，定下优先顺序；三是进一步拆解流程，仔细分析各个步骤，看看问题可能出在哪里。

（2）D（Do）：指根据计划动手改进。需要事先想好退出计划，在动手时应从小处切入，谨防破坏现有流程，造成系统瘫痪。如果解决方案不止一个，就应该设计一个实验，评估各方案的优劣。

（3）C（Check）：指监控改进过程。需要确认测量工具和方法维持不变，否则可能误判。在结果数据出来后，需要仔细分析，看看哪种方案比较有效，而且对原来流程的变更幅度最小。

（4）A（Act）：指确认改善的确有效，在不良和缺陷已经消除后，导入及固化流程，改写规范。若改善无效，则检讨原因、修改计划，重新启动一轮 PDCA 循环。

PDCA 循环是爬楼梯上升式的循环，每转动一周，质量就提高一步。在一轮循环结束后，也许只解决了一部分问题，或者在解决的过程中又衍生出新的问题，因此必须再进行一轮 PDCA 循环。如此循环迭代，不断提高软件的质量。

对于整个软件开发生命周期而言，软件质量贯穿于各个环节当中，从需求分析开始，一直到发布上线，PDCA 循环则是项目迭代顺利完成的重要保障，如图 6.3 所示。

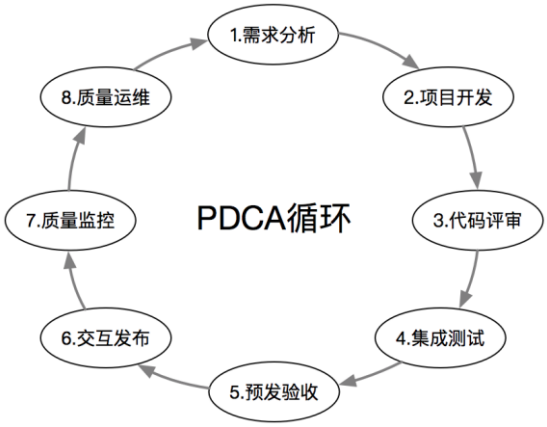


图 6.3

6.2　从黑天鹅事件到墨菲定律

软件系统的稳定性，主要取决于整体的系统架构设计，然而也不可忽略编程的细节，所谓“千里之堤，毁于蚁穴”，一旦考虑不周，看似无关紧要的代码片段便可能导致软件系统的整体崩溃。

本节主要介绍几个重要的理论，分别是黑天鹅事件、蝴蝶效应和墨菲定律。这些理论在软件开发领域尤其是质量方面都至关重要。在了解这几个理论后，我们会发现，保证软件系统的质量其实并不容易。

6.2.1　黑天鹅事件

黑天鹅寓意不可预测的重大稀有事件，它在意料之外，却又改变着一切。人类总是过度相信经验，殊不知其实一只“黑天鹅”的出现就足以颠覆一切。所以，黑天鹅事件一般指难以预测且不寻常的通常会引起市场连锁负面反应甚至颠覆的事件。泰坦尼克号沉没、9·11 事件、美国的次贷危机、2008 年中国的雪灾，都是比较典型的黑天鹅事件。

黑天鹅事件通常具备如下三个特点：

（1）稀缺、史无前例（Rarity）；

（2）影响很极端（Extreme Impact）；

（3）具有意外性，但人的本性促使我们在事后为它的发生编造理由，并且或多或少认为它是可解释和可预测的。

“黑天鹅”存在于各个领域，无论是金融市场、商业、经济还是个人生活，都有它的踪迹。最近几年发生的几个 IT 领域相关的黑天鹅事件如下。

◎ 2015 年 5 月 27 日下午 5 点左右，某支付平台出现网络故障，导致账号无法登录，无法完成线上支付。后经证实，该故障是由于杭州市萧山区某地光纤被挖断导致的，这一事件造成部分用户无法使用该支付平台，其影响辐射到全国范围内的各个领域。

◎ 2016 年 3 月 15 日，Google DeepMind 公司开发的围棋人工智能程序 AlphaGO 和韩国著名的围棋专业选手李世石进行了对决。最终双方总比分为 4:1，AlphaGo 取得了人机围棋对决的胜利。2017 年 5 月 23 日，AlphaGo 再次与人类围棋高手柯洁展开对决，最终柯洁中盘投子认输。赛后柯洁表示：“它太完美，我看不到希望。”

◎ 2017 年 5 月，一种名“WannaCry”的蠕虫式勒索病毒利用 NSA（National Security Agency，美国国家安全局）泄露的危险漏洞 EternalBlue（永恒之蓝）进行传播，全球 150 多个国家和地区超过 30 万台电脑遭到了勒索病毒攻击，影响到金融、能源、医疗、教育等众多行业，损失高达 80 亿美元。

◎ 2018 年 6 月 10 日，韩国交易所 Coinrail 宣布其系统检测到黑客攻击，NPXS 已经受损，被盗虚拟货币的价值达到 400 亿韩元（约人民币 2.4 亿元）。CoinMarketCap 的数据显示，从交易量来看，Coinrail 刚排进全球前 100 名，但是其被黑客攻击所导致的影响是非常巨大的。在 Coinrail 遭受攻击后的 24 小时内，市值前 100 的加密货币有 99 种价格下跌，总市值缩水约 316 亿美元，首当其冲的比特币跌破 7000 美元，比特币的价格在三个交易日下跌了近 12%，引发了价值 460 亿美元的抛售，并将自 2018 以来比特币的跌幅扩大至 50%以上。

近三年来，在 IT 领域比较大的 4 起黑天鹅事件中，有 3 起是因为软件系统的可用性、安全性等受到挑战导致的，后果相当严重。

英国著名哲学家弗朗西斯·培根就曾发出这样的警告：我们要当心被自己思想的丝线束缚。无论是泰坦尼克的沉没，还是中兴被制裁，如果业界没有类似的案例，那么几乎无从借鉴。即使有业界案例，不同的组织、公司未必有相应的处理经验，所以自己的思想、经验也是非常有局限性的。有时我们太看重自己知道的事，却没发现我们不知道的事比我们知道的事更有意义，所以只有反常地思考一切，才有可能发现更多的“我们不知道的事”。

6.2.2　蝴蝶效应

20 世纪 70 年代，美国气象学家洛伦兹在解释空气系统理论时说：“亚马孙雨林中的一只蝴蝶偶尔振动翅膀，也许会引起两周后美国得克萨斯州的一场龙卷风。”

这就是著名的蝴蝶效应，意思是初始条件十分微小的变化经过不断放大，也许会对未来造成极大的影响。

图灵奖得主 Tony Hoare 在 *Null References: The Billion Dollar Mistake* 文章中写到：“我将 Null 引用称为自己的 10 亿美元错误，它的发明是在 1965 年，那时我用一种面向对象语言（ALGOL W）设计了第 1 个全面的引用类型的系统，目的是确保所有引用的使用都是绝对安全的，编译器会自动进行检查。但是我未能抵御住诱惑，加入了 Null 引用，仅仅因为实现起来非常容易。Null 引用导致了数不清的错误、漏洞和系统崩溃，可能会在之后 40 年里造成 10 亿美元的损失。”

Null 引用在 Java 中的表现就是可能抛出 NullPointerException 异常，这是一种非预期的异常。对于 IT 系统而言，还有很多非预期的错误及异常，都可能是蝴蝶效应引起的大故障，例如非预期 error、非预期的调用抖动、极少数场景下的规则未被正确处理、错误的优惠处理逻辑、未正确设置的营销活动，等等。

如果不具备快速、智能的感知能力，那么可能影响更多的用户，使资金损失增加、业务不可用时间变长，等等。

6.2.3　墨菲定律

“墨菲定律”是一种心理学效应，由爱德华·墨菲（Edward A. Murphy）提出。它的原句描述为：如果有两种或两种以上的方式去做某件事情，而其中一种选择方式将导致灾难，则必定有人做出这种选择。

墨菲定律（Murphy’s Law）的主要内容如下：

◎ 任何事都没有表面看起来那么简单；
◎ 做所有的事花费的时间都会比你预期的时间长；
◎ 会出错的事总会出错；
◎ 如果你担心某种情况会发生，它就更有可能发生。

爱德华·墨菲是美国爱德华兹空军基地的上尉工程师。1949 年，他和他的上司斯塔普

少校参加美国空军进行的 MX981 火箭减速超重实验。这个实验用于测定人类对加速度的承受极限。其中有一个实验项目是将 16 个火箭加速度计悬空装置在受试者上方，当时有两种方法可以将加速度计固定在支架上，而不可思议的是，竟然有人有条不紊地将 16 个加速度计全部装在错误的位置。于是墨菲做出了这一著名的论断，意思是如果做某项工作有多种方法，而其中有一种方法将导致事故，那么一定有人按这种方法去做。

在 IT 系统中有很多关于墨菲定律的案例，接下来会列举其中的两个案例。

这里列举第 1 个案例。笔者曾遇到一位经验丰富的架构师，他对系统迁移过程中的自校验、核对、切流策略、灰度能力、回滚机制、容错处理等都进行了充分考虑，但未充分考虑老系统的一种流程处理缺陷的备案或者处理方案，因为老系统在上线的一年内只发生了 1 起相关故障。但该缺陷的影响在新系统中被无限放大，很快就导致 1 起严重的线上故障。

这里列举第 2 个案例。很久以前，笔者所在的公司来了一位专家级别的开发人员，他面对的第 1 个小需求就是把搜索结果列表从每页 10 条改为每页 50 条。他认为只改一个分页的 pageSize 就可以了，于是很快就改完了，并且没经过测试就发布到线上，结果导致公司整个网站的搜索结果跌零。后来经排查发现，因为老系统的性能不高，所以开发人员做了一个设置：当搜索结果不少于 50 条的时候，就直接返回 0 条搜索结果。暂且不论这种设置是否合理，这种故障之所以发生，就是因为开发人员过于自信，疏忽了重要的测试环节。所以，在软件测试方面，没有经过测试的功能往往会出问题。

由于认知的局限性、骄傲心态、问题域的复杂性及不可把握性等因素，软件行业发生问题的概率很高。墨菲定律告诉我们：事情往往会朝着我们所能想到的不好的方向发展，只要存在这种可能性。所以，无论是开发人员还是测试人员，都需要谨记墨菲定律，莫存侥幸心理，要把自己的系统设计得足够强壮。

6.3 软件质量稳定性之殇

黑天鹅事件告诉我们要走出经验主义，不要将自己知道的东西太当回事，我们不知道的事比我们知道的事更有意义。蝴蝶效应告诉我们要及时感知问题，止损和熔断，避免问题范围扩大甚至传染到其他相关领域。墨菲定律告诉我们，我们知道的但是忽略的地方终究会出问题。

一言以蔽之，要将软件研发中的质量搞好并非易事。导致软件质量问题难以把控的原

因还有很多，本节会详细进行分析。

6.3.1　业务高速发展带来的变化

在很多年前就有人想将软件从业者变成制造工人，不断进行流水线工作。但这几乎没什么可能，因为软件从业者要解决的问题域太复杂。虽然业界有很多规范、标准、套装软件，但是仍然未解决问题的万分之一。

阿里巴巴的电商业务在最近几年呈指数级增长，业务的发展速度惊人。阿里巴巴内部有一句话说：双 11 是业务的狂欢、技术的盛宴。到目前为止，阿里巴巴做了 10 年双 11，双 11 交易额从 2009 年的 5.9 亿元增长到 2018 年的 2135 亿元，实现了 355 倍的增长，交易峰值也从 2009 年的 400 笔每秒，增长到 2018 的 49.1 万笔每秒，系统规模也有几十倍、上百倍的增长，如图 6.4 所示。

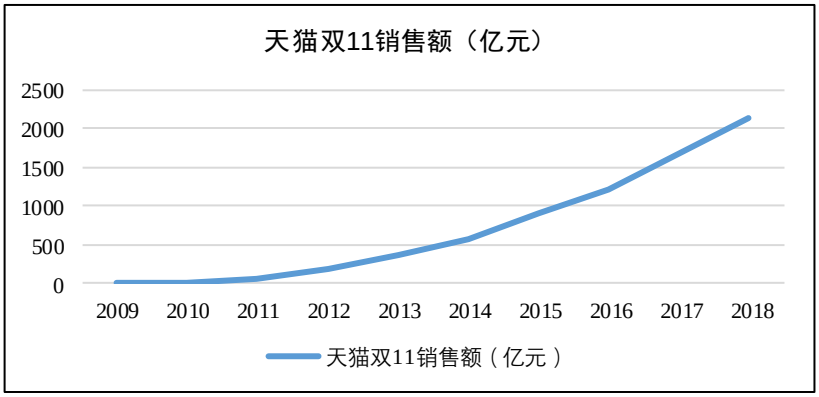


图 6.4

随着业务的高速发展，技术挑战越来越大，有些挑战甚至是无法借鉴的，只能靠自我创新。阿里巴巴面对的双 11 技术挑战，可以说是世界性的，特别是如何在零点峰值到来时确保系统稳定。

6.3.2　问题域的复杂性

IBM 大型机之父佛瑞德·布鲁克斯在《没有银弹：软件工程的本质性与附属性工作》论文中提到：软件开发是极其困难的，这些困难分为两类：本质性的和附属性的；附加性

的困难会随着工具的改善逐渐淡化，本质性的困难是最难解决的，其要点是问题域的复杂性。

问题域，就是针对现实世界中某个特定问题的所有相关方面的抽象。现实世界中的某些特定问题本来就是极其复杂的，根据这些问题做的抽象，问题域自然不会简单。因为不同的人看问题的角度可能不一样，所以抽象出的结果可能大不相同。

6.3.3 系统的复杂性

随着业务的发展，人们对可用性的诉求越来越高，但是可用性的提高必然导致系统的复杂度攀升。为了提升系统的可用性，应用的部署就可能从单机部署演化成集群部署。而集群部署在初期可能采用同城多机房的容灾方案，慢慢又会演化成异地多机房的容灾方案。而随着方案越来越复杂，多个机房之间的负载均衡、容灾方案等挑战也会越来越大。这还只是考虑了线上生产环境中的机器情况，再加入测试环境、灰度环境、预发布环境等，就会更加复杂。

除了集群部署，为了提升可用性等问题，我们也会将单体应用拆分成分布式应用，进而演化成微服务，各个应用之间通过 RPC、异步消息等方式进行通信，这就会引入更多的数据一致性等问题。

举个例子，有一个主要用于支付链路的规则决策的系统，该系统起初可能只有 4 万行代码，后来增加到 8 万行代码，现在又增加到 10 万行代码。代码行增加了，该应用的功能也增加了，同时系统的调用逻辑及运算的复杂度必然增加，那么如何保持对外 API 的 TPS 不降低，RT 不降低？

6.3.4 技术债问题

技术债务是由 Ward Cunningham 在 1992 年创造的一个比喻，被定义为我们有意或无意中做了错误的或不理想的技术决策所累积的债务。

Steve McConnell 将技术债务分为如下两类。

◎ 无意的：由于经验的缺乏导致初级开发者编写了质量低劣的代码。
◎ 有意的：团队根据当前而非未来进行了设计选型，这种方式可能很快解决了当前的问题，但很拙劣。

随着技术团队人员的能力提升，无意的技术债务可能会越来越少。但是在很多时候，我们欠下的技术债务往往是有意的，随着业务的高速发展，还会带来很多变化。为了快速实现业务的功能及验证商业决策，开发人员大多会采用很多临时方案。毫不客气地说，这些临时方案其实就是技术债。

临时方案的可怕之处，不仅仅在于其不全面，更在于如果不对其及时处理，可能就永远“临时”下去了。随着团队成员的不断更替，慢慢地就没人记得这个临时方案了。一个说好只支撑两个月的临时代码，可能两年后还在线上机器里运行着。

随着系统中的临时方案越来越多，我们需要还的技术债也越来越多。欠债并不可怕，可怕的是没有偿还计划。其实，任何系统都无法避免技术债，因为业务在不断发展，总需要做些妥协，这都无可厚非，但是一定要注意如下几点。

◎ 通过人员培训、代码审查等方式杜绝无意的技术债。
◎ 要尽量避免有意的技术债。
◎ 如果一定要引入技术债，则一定不要引入不计后果的技术债。
◎ 要记录所有欠下的技术债，并且有偿还计划。

技术债务和金融债务非常相似。一个人贷了款就会产生债务，如果可以定期还款，则所欠的债务是可以接受的，不会产生进一步的问题；但是，如果不还款，就会以利息作为惩罚，利息还会随着不还款次数的增加而增加。如果这个人在很长一段时间内不能支付任何款项，那么应计利息会使他更难以偿还债务。在极端情况下，这个人就不得不宣布自己破产。技术债务也一样，如果越积越多，最终也会导致非常严重的后果。

6.3.5　人、流程、文档的博弈

面对复杂的问题域，人们都有简单化思考的倾向。比如某开发人员将手工修改的代码通过自动生成工具覆盖了，这时会有两种选择，一个是通过两个目录来区分，一个是开发一款检查工具来做合并前的审核。这些做法都没有问题，但是不要靠脑子去记。

对于一个存在 3 年以上的产品，当有新人到来的时候，他们总会在第 1 个月提这样的问题：“这个产品怎么采用这样的技术啊？理解成本这么高，一点文档都没有！只能看代码，代码写得还不怎么样……”当新人成了“老人”，他们会对更新的新人说：“我们当年更惨，只有看代码是靠谱的。对了，有某位专家非常了解相关内容，有问题找他，比看文档有用。”

总之，我们不得不承认的事实包括：

◎ 无论写多少文档，如何维护“活文档”（一个特别熟悉代码的人）都是一个大问题；
◎ 关键时刻要靠人传、帮、带，要靠传承。

6.3.6 采用不能掌控的工具和框架

技术人员大多有采用新工具、新语言、新开源产品的追求。当全栈、多语言风潮席卷各社区的时候，不罗列一堆名词真心不好意思跟人交流。这还涉及技术带头人的偏好问题。

在前面的章节中，我们鼓励开发者学习新技术、新语言等。但是不得不说，有很多线上故障都是由于开发者对所使用的新技术不了解导致的。所以，对于新技术、新语言，我们应该充满期待，但是在真正将其引入日常工作中的时候，一定要考虑整个团队对于这项技术的接受程度及掌握程度。

6.3.7 质量意识

经过分析，80%的线上问题并不那么难以解决，其本质是未遵循规定的动作。比如修改了代码且未充分验证，认为就修改了几行代码，咋看都不会出错。从内建质量的角度来看，我们一般有多条质量防线，一个问题被遗留到线上，往往有 3 个以上的措施都失守了。很多故障的发生，都是因为质量意识的薄弱导致的。

无论是开发人员还是测试人员，都应该对线上环境保持敬畏之心。前面提到过墨菲定律、蝴蝶效应及黑天鹅事件等，都是希望提醒读者们，尤其是开发者，要对自己的每一行代码都保持敬畏之心。

有一篇关于如何规范飞行员作风的文章提到的如下经验颇有借鉴意义。

◎ 不忘初心，严格遵守 SOP（标准操作程序）。
◎ 减少侥幸心理，增强风险意识。
◎ 狠抓作风养成。
◎ 严厉处罚无后果违章，既要结果，又要过程正确。

6.4　从康威定律和技术债看研发之痛

本节在技术债的基础上，再介绍一个比较重要的定律——康威定律，然后从康威定律和技术债的角度看研发之痛。

6.4.1　康威定律

康威定律由 Conway 在 1967 提出，其主要观点是：设计系统的架构受制于产生这些设计的组织的沟通结构。

其实，这里的系统并不局限于软件领域的系统。康威对其定律又做了如下具体解读。

◎ 组织沟通方式会通过系统设计表达出来。
◎ 时间再多，对一件事也不可能做得完美，但总有时间做完。
◎ 在线型系统和线型组织架构间有潜在的异质同态特性。
◎ 大系统的组织总是比小系统更倾向于分解。

也就是说，需要构建什么样的系统，就搭建什么样的组织结构。

6.4.2　通过 3 个案例来看研发之痛

下面结合案例来深入理解康威定律。

1. 接入效率优化

以真实的某个业务为例，在 2018 年年中的时候，我们负责的一个产品需要接入一个全新的业务模式，经过初步评估，整个过程多达 10 天，历经 17 个环节，涉及 8 个团队，如图 6.5 所示，我们期望在此过程中提升整体效率。

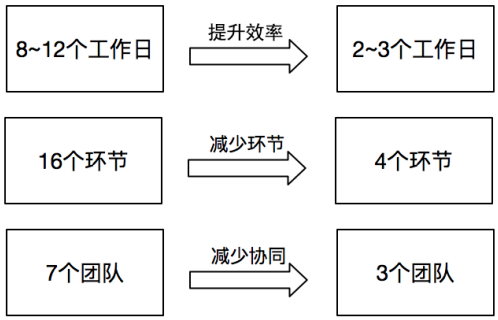


图 6.5

面对以上情况，我们确定了几个策略以期带来改变：减少协同、减少环节、提升效率。

为什么要减少协同呢？前面讲过，组织沟通方式会通过系统设计表达出来，沟通成本随着团队成员的增加以几何级数增加。关于沟通成本，在《人月神话》中给出了很简洁的答案：沟通成本 $= n(n-1)/2$，n=项目成员。如果按一个团队 1 个成员作为接口人参与跨团队协同，那么 8 个团队的沟通就可能达到 87/2=28；而 3 个团队的沟通值算下来是 32/2=3。

为什么要减少环节呢？环节越多，涉及的组织和团队就越多，减少组织协同必然要减少环节。流程越烦琐，就越会影响办事效率，软件研发亦是如此。

减少了环节，减少了协同，效率自然就高了。那么有哪些减少协同的方式呢？

我们来看一些例子。如图 6.6 所示，如果一个团队有 3 个团队在协作，而每个团队都是前端、后端、测试职责分明，那么需要协同的环节必然很多。团队 1、团队 2 和团队 3 都会分别进行团队内部的前后端开发联调、交付测试及验收测试等，且相互之间需要进行联调及联测。

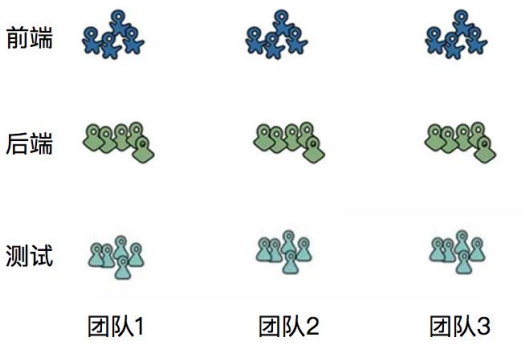


图 6.6

如果对这 3 个团队进行功能合并，就会形成全栈型团队，可能变成如图 6.7 所示。



图 6.7

全栈型团队虽然也有编写代码、单元测试、接口测试、集成测试等环节，但跨越组织的沟通已经减少了很多。

如图 6.8 所示，我们再尝试一种变化，因为设计系统的组织等于组织内和组织间的沟通结果。

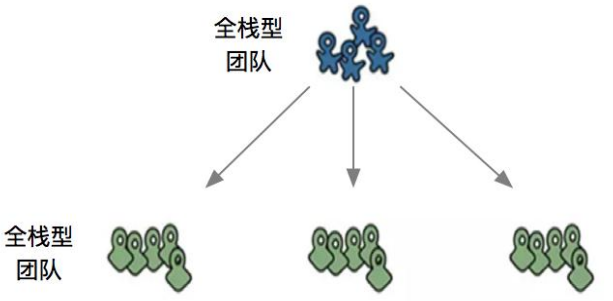


图 6.8

这个组织沟通图背后的架构关系可以如图 6.9 所示。

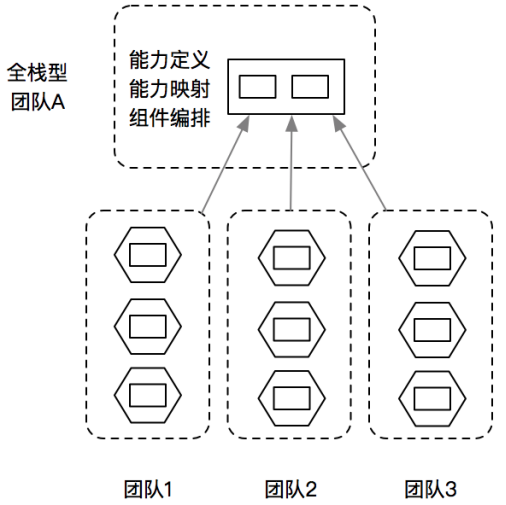


图 6.9

总之，组织关系和架构息息相关，太多的沟通成本往往由于组织协同方太多造成，应该从外部视角来组合组织关系，尝试通过抽象、构建通用组件来达到能力复用，尝试对用户或者内部客户提供统一的受理界面。

2. 重复的轮子

在软件开发领域有一个流行的原则：DRY（Don't Repeat Yourself），我们一般将其翻译为：不要重复造轮子。

这里抛开其他，聊聊轮子（或称产品）与组织的关系。首先，如果在同一个领导下已经有一个轮子了，那么兄弟团队用还是不用这个轮子？如果不用，则可能无法过领导那一关，尽管你不喜欢用这个轮子，却还是得用。

如果是不同组织的轮子呢？就不一定了。如果是一个业务研发型团队，则可能没有时间去造中间件轮子，但是可以搞些 common-tools 之类的轮子，这些也可以作为创新产出去交差。而且我们可能会对同类轮子采取怀疑态度，会想办法证明自己的轮子更好。当然，另一个团队也是这样向领导汇报的。最终，这样的轮子越来越多且很难推广，也无法解决全部的问题，因为问题域很广，没有银弹。

这里有个案例，有三个业务团队都需要一套监控系统，已知目前有一些相对成熟的产品，但是都无法满足自己的个性化需求，如图 6.10 所示。

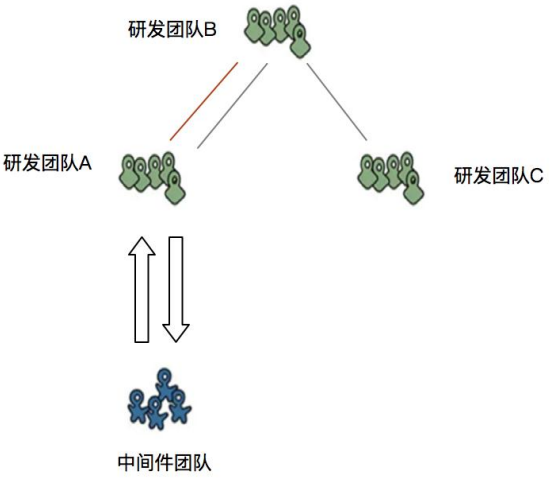


图 6.10

在这个案例中，关于各个团队如何实现自己的定制化监控需求，有如下 3 个方案。

◎ 方案 1：向基础平台提需求。
◎ 方案 2：自己做能力升级。
◎ 方案 3：找找别的轮子。

方案 1 是最好的，专业的事情交给专业团队来做；方案 2 是不得已而为之的；方案 3 是折中的。也就是说，方案 3 优于方案 2，但我们习惯于做方案 2，造一个轮子，因此形成鄙视链，如图 6.11 所示。

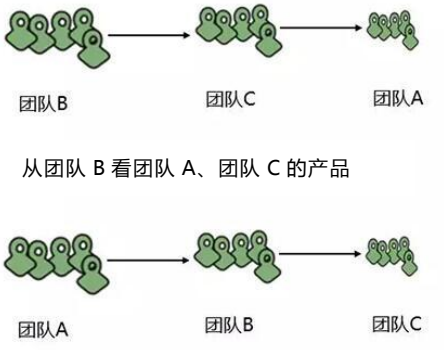


图 6.11

“你需要搭建什么样的系统，就需要搭建什么样的组织结构”这句话有一种逆向作用力。为了满足 *X* 领域的监控能力，我们在基础平台的基础上引入了 *X* 实时计算、*X* 指标管理及 *X* 模型，还成立了一个业务团队 A 来做这件事，进而形成一种认知：业务团队 A 就是来解决这个问题的，而且很强大，业务团队 A 致力于解决更多的类似问题。后来发现业务团队 B、业务团队 C 也是通过自然生长解决类似问题的。那么问题来了，从公司的角度来看，业务团队 A、B、C 做的事情应该合并；从各团队的角度来看，他们会不断证明别的团队做的产品并无法满足自己的需求，自己才是应该留下来的团队。

总之，你要搭建什么样的系统，需要搭建什么样的组织结构，一旦搭建了这样的组织结构，就可能带来逆向作用力，从而导致重复建设、效率低下、部门墙，且最终可能没有打穿任意一个问题域，还为合作伙伴或者客户提供了非一站式的解决方案，陷入各自为政的矛盾中。

在提出解决办法前，这里先抛出一些原则：协同、赋能、动态组织。

首先，明确一个动态组织的概念。组织是为业务或者某些目标服务的，应该具备灵活性、动态性。一个长期稳定的组织可能没有活力，或者没有新的挑战空间。

其次，协同原则。一个组织要明确自己的目标，并明确有多少事是通过和协作方协同合作共同完成的，否则会让团队泥足深陷。赋能，是基于协同这个共识而来的，中间件团队要赋能业务研发团队自不必说，业务研发团队之间也可以相互赋能，对于共性问题，没有必要重复造轮子。这里引入了一个非常关键的因素，就是考核！业务团队 A 的主管在考核业务团队 A 的绩效目标时，将自己的团队做出哪些工具作为目标是不正确的，业务团队 A 的主管的主管的目标可能是达成 5 秒的异常检测能力，并不关注监控的一部分能力是谁提供的。所以，基于以上内容，可以给出一些解法，但不限于此。

如图 6.12 所示，中间件团队可以将监控平台做成内部开源模式，暂不讨论其是完全开放的，还是提供基础能力层的，其他业务团队都可以贡献代码，业务团队的优势是离业务场景很近，有具体的场景驱动。这样还可以解决业务团队 A、B、C 的优先级问题，如果全部把需求提交给中间件团队，则完成的时间就成了未知数。

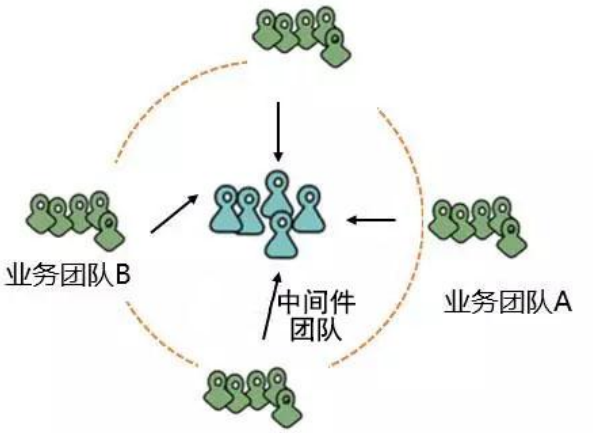


图 6.12

如图 6.13 所示，业务团队 A 和中间件团队可以持续共建某些基础能力，可能业务团队 A 是需要赋能的大户，而且需要的成本很高，所以长期共建。基于以上协同、赋能、动态组织原则，业务团队 A 在完成使命后可以在对应的组织内重新确定目标，同时，为业务团队 B、C 服务。对于业务团队 B 而言，可能只需要投入 1 ~ 2 个人，做上两个月就可以满足自己的需求了。

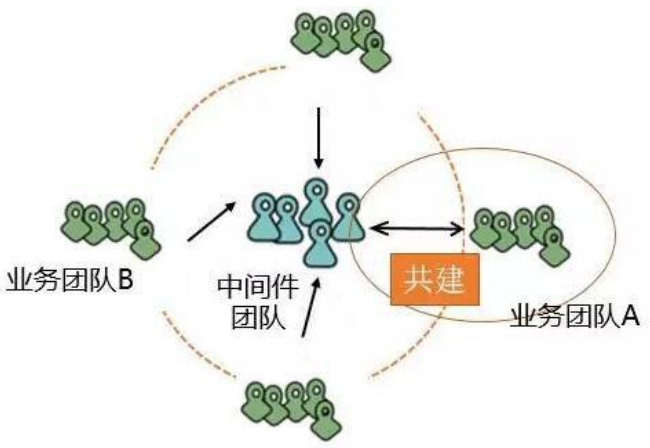


图 6.13

3. 平台发展 vs 业务发展

软件行业有很多大咖提出：架构是演进出来的，而不是设计出来的。笔者比较赞同这种说法。总体来讲，演进是必需的，前期设计做多少合适是值得探讨的。一个衡量标准是对于方案 A、B 进行评估，如果方案 B 可以更通用，付出的成本也在可接受的范围内，则优先选择方案 B。不做 BDUF（Big Design Up Front，臃肿的预先设计）是大家的共识，而艺术在于对“度”的把握。

许多人都会面临一个问题：是先发展平台，还是先发展业务？按理来说，平台应该随着业务一起发展，但是如果每个需求都堆砌式上线，就构建了一个个的烟囱。我们期望在一段时间后，相似度高的同类型需求可以有快得多的上线能力。

关于平台发展和业务发展，以及不同场景、不同阶段的业务架构原则，这里给出几个观点。

◎ 架构方案和业务形态息息相关。
◎ 架构是演进的、发展的，没有最好，只有适合。
◎ 技术债要做清偿计划。
◎ 不要在下雨天补屋顶。

6.4.3 架构方案和业务形态息息相关

笔者之前经历了一条产品线，完全是试错型的产品线。我们在设计之初对模型的共同层做了抽象，后来五花八门的产品长在上面，公共的内容越来越少，反而导致了系统调用链路增加，以及代码编写的复杂度增加。1 年后，这条产品线全线关闭。从这个案例来看，最适合的就是满足业务快速试错，可能烟囱式架构是最合适的，甚至产品之间有一些代码重复，但各团队各司其职，是最快反应的最小研发响应单元。对应到康威定律的组织和架构关系，这也是依赖最少的组织，沟通成本最低。

如图 6.14 所示，烟囱型架构在不确定产品线的试错方面有优势，对应的组织构架也应以简单、快速响应为特征。

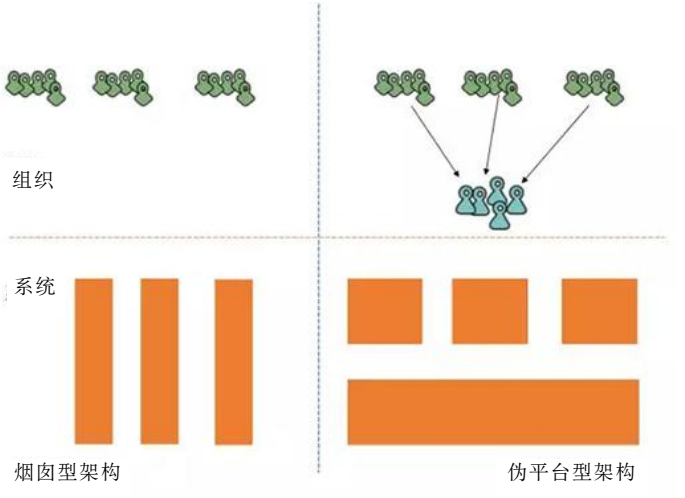


图 6.14

这里再举一个例子来说说架构演进这事。如图 6.15 所示，随着业务的发展，类似的诉求会凸显，因此沉淀为越来越多的平台。

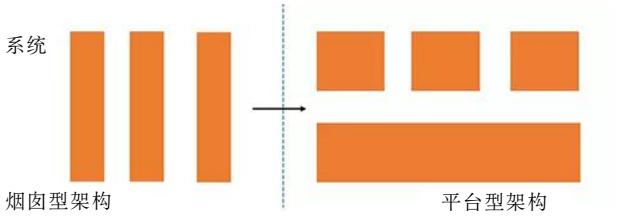


图 6.15

其中，除了有需求的相似性、平台要求的相似性导致的架构发展，还可能有对领域本质认知的升级。比如在支付宝发展过程中先后使用过红包、实时优惠、商户优惠券等产品，这些都是烟囱式架构发展下的产物。如图 6.16 所示为一个红包产品的展示。



图 6.16

行业也有其他类似“券”的产物，例如京东红包、苏宁红包、滴滴打车券、滴滴打车红包、微信红包、电信充值红包、大众点评红包等。

简单来看，券和红包是两类东西。但是，我们在产品上形成了如下定义：如果可以视作一类东西，则可以对它们进行统一的抽象和管理。

券是一种票据，作为券发行方和拥有方之间的凭证，具有一定的价值和法律效应。

相关干系方如下：

◎ 券的发放方（提供权益）；
◎ 券的拥有方（享受权益）；
◎ 券的发放工具（是券的发行方向券的拥有方发放券凭证的工具）。

券的形式如下：

◎ 可以将券以介质分类为纸质券、电子券；
◎ 可以将券以使用方式分类为入场券、礼品券、提货券、代金券、红包、打折卡、满减卡等；
◎ 可以将券作为基础产品，在业务形态上包装为打折卡、满减卡等用户可感知的产品或者营销工具。

6.5 求解质量熵

在前面的小节中，我们从黑天鹅事件谈到了蝴蝶效应和墨菲定律。一言以蔽之，将软

件研发质量做好并非易事。质量是一个综合命题，涉及业务的准确性、稳定性和可用性等，比如某厂商在大促的几个小时内无法交易、收藏夹的商品丢失、用户享受的优惠不符合预期、营销规则难以解释等。

黑天鹅告诉我们，要走出经验主义，就不要将自己知道的太当回事，我们不知道的事比我们知道的事更有意义；蝴蝶效应告诉我们，要及时感知问题、止损和熔断，避免问题范围扩大甚至影响到其他相关领域；墨菲定律告诉我们，你知道却忽略的问题终会爆发！

在前面的小节中还讲解了让人难以掌控质量问题的复杂因素，如下所述。

◎ 业务高速发展带来的变化。
◎ 技术债。
◎ 人、流程和文档的博弈。
◎ 采用了不能掌控的工具和框架。
◎ 复杂的问题域。
◎ 质量意识。

为了应对以上复杂因素，这里从如下几方面来思考对策。但是，如同没有银弹一样，也没有万能解药。

6.5.1　运用敏捷思想

敏捷开发思想以用户的需求进化为核心，迭代式、循序渐进地进行软件开发。在敏捷宣言中是这样说的：我们一直在实践中探寻更好的软件开发方法，在身体力行的同时帮助他人。由此，我们可以建立如下价值观：

◎ 个体和互动优于流程和工具。
◎ 能工作的软件优于详尽的文档。
◎ 客户合作优于合同谈判。
◎ 响应变化优于遵循计划。

敏捷建模（Agile Modeling，AM）的价值观包含了极限编程（Extreme Programming，EP）的 4 个价值观：沟通、简单、反馈、勇气，还增加了 1 个价值观：谦逊。至少，我们明白了快速反馈的好处：做错了、做偏了可以马上校正。瀑布模型式的开发反馈必然是慢的，一位做医疗单机软件的专家说，他们半年才进行一次新版本发布，可见其中的提升空间有多大。

总结一下：基于敏捷的思想可以快速发布、纠正和调整，以满足用户的需求。

6.5.2 运用系统化思想

系统化思想就是通过类比、抽象、层次化等手段挖掘本质。很多时候，当软件系统在线上出现 Bug 时，我们会开展各种故障复盘并定下很多后续的行动点，但下次依旧会犯同样的错误。究其本质，一方面是我们将复盘的内容聚焦在单个事件上，对于更大体系的影响洞察不够；另一方面是未持续学习，不情愿承担问题。

但是，建立系统化思想其实是比较难的，太多人对建立新思想持抵抗态度。如萨提亚改变模式（见图 6.17）所示，当我们适应了原有状态却需要面对改变时，我们会经历抵抗、混乱、改变主意、重新上路等过程，最后适应新的状态。

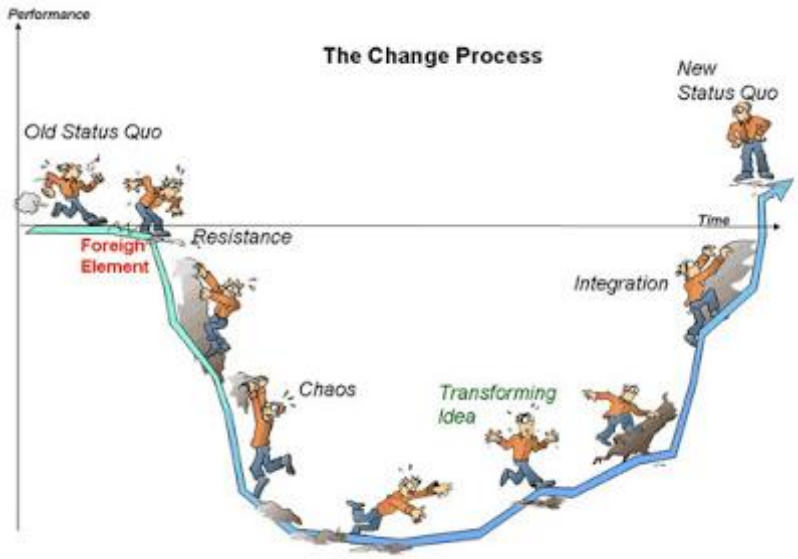


图 6.17

之前，笔者所在的团队对深度复盘实践也是比较抵触的，通过大约 1 年的持续练习，才使我们对于问题或故障的认知及担当力都上了一个台阶。

6.5.3　技术债偿还计划

在 6.4 节提到，技术债的存在是软件质量无法得到很好提升的一个重要原因。对于遗留的技术债，很多开发人员很想解决，但是由于各种各样的原因，技术债越积越多。

关于技术债的偿还有如下 4 个有趣的观点。

◎ 技术人员水平不行、意识不够，代码越写越乱。
◎ 开发者都有一颗积极向上的心，但不知道如何做才好。
◎ 上行下效，没有靠谱的 Leader、架构师，大家便都是懒惰的，不想去还债。
◎ 业务压力太大，赶着交付，没有时间清偿技术债。

这里总结了几个有效应对思路。

◎ 构建质量保障体系，让开发人员有勇气动手改造。比如进行接口测试、单元测试，通过持续集成进行反馈，这样开发人员在做重构的时候就有一些基本保障。
◎ 不要动遗留系统中没有出问题的地方，对于遗留系统中出问题的地方，修复并补充测试代码，更进一步地，对高危功能和模块做定向增强。
◎ 招募优秀的工程师，使好的工作作风和习惯影响整个团队。
◎ 抓住痛点大做文章。比如，平时说持续集成，可能团队成员没有感受到其好处，那么出 Bug 了再讨论讨论；复制粘贴来的代码总有不能满足更复杂需求的时候，这时讨论，大家就会发自心底地认同，觉得必须得重写代码了。改变人很难，但这样的机会不容错过。一定要抓住这样的机会，构建品质高一些的代码和质量防线。

6.5.4　抓住合适的时机做架构升级

在做软件架构升级的时候，一定要考虑时机是否合适。当一些重点战役型项目让你喘不过气的时候，去做理想架构肯定会碰壁。这里给出的一个观点就是技术、架构始终为业务服务，为商业服务。无论是从走出舒适区，还是从架构的发展来讲，一定要找准时机做架构升级，这个升级要基于对业务本质的不断认知，否则就会南辕北辙。

在一次重要的架构改造过程中，笔者所在的团队对新的技术架构方案的确定比新业务的来临早了 3 个月，这样刚好通过一个小的业务需求使新的技术架构快速上线。新的技术架构刚好又在之后的新业务需求中派上用场，帮助我们快速地将商业项目落地。可见，3 个月的时差足以改变许多东西。

因此要记住，要抓住合适的时机做架构升级。

总结一下：康威定律及其推论告诉我们要做什么事情，以及设定什么样的组织结构；同时做不同的事情，要允许有动态、弹性的组织结构，不要被组织束缚；更进一步，通过跨越组织、立足协同及赋能可以有更大的组织成效；技术债务不要等到“下雨天”才偿还，平衡做业务和平台架构发展的艺术取决于一定时间和环境下的目标，技术人员不要永远处于被驱动的状态！研发之痛，是一个复杂的调试性问题，没有银弹！

6.5.5 内建质量

前面讲技术债务和反馈的时候都提到了持续集成，从整体来讲，内建质量就是用更短的时间构建，以及用更快的速度反馈。

同时，除了正确做事，还要思辨“做正确的事”。通过 ATDD、引流对比、串讲反串讲、评审、录制回放验证、接口回归测试等手段可以验证从需求到设计再到实现是否发生了变形。

内建质量=正确做事 × 做正确的事。

6.5.6 不迷信，不唯新、不唯上，实践是检验真理的标准

在技术选型上，我们要不迷信、不唯新、不唯上，建议尽早做小范围的可行性论证。比如对于之前流行的 OSGi 技术，采用它可以获得什么，其风险又是什么。

在引入一种新的技术栈的同时，会带来一层复杂度，而复杂度有一个天性：只能增加，不能减少，就像宇宙中的熵一样。比如，一旦开放使用某个接口，就很难将这个接口停用。一旦以某种高耦合方式引入某种技术，调整或者去除该技术的成本就会呈几何级数形式。

总之，在做技术选型时，要看看技术的成熟度、团队人员的掌握程度、收益和风险评估，一旦陷入泥潭则无异于梦魇之旅。为了避免陷入泥潭，我们可以基于风险的架构设计，采取可行性论证。

6.5.7 复杂的问题域：专项突破

前面提到，系统和问题域的复杂度都会增加出现 Bug 的风险。

随着业务的发展，人们对可用性的诉求也越来越高。比如从单机房部署演变成同城异机房，最后又演变为异地机房。因为有异地机房，所以 RPC 的路由复杂度增加；同样，在开发环境、线下测试环境下模拟几种部署方式也是有复杂度的。

对于这样一个庞大的系统来说，在每次发布新版本时不仅要完成功能用例的构建，还要完成性能的测试。

那么，面对如此复杂的测试，我们该如何应对呢？答案是，除了需要构建功能回归测试环境，还需要构建性能回归测试环境，以保障每次发布的产品性能是没有降低的。

对于多机房部署这种情况而言，在技术架构上不但要考虑复杂度带来的诸多依赖产品的修改成本，还要考虑易用性和如何测试，甚至如何方便、低侵入地测试。因为一旦从单机房部署演变为多机房部署，问题的外延就翻倍了。比如：一个定时任务会不会同时在两个机房运行并导致数据重复；在机房切换时是否有缓存切换的问题导致业务不可用；数据复制延迟带来的不一致性对业务的影响；对数据复制的监控、对缓存的监控等，需要有全局的唯一性管理策略及业务兼容性方案，等等。

6.5.8　领导者的意识

一个公司的文化由创始人决定，一个团队的品质追求也往往由其领导者决定。领导者不自觉的保护主义及报喜不报忧等做法都会让问题变得严重。

团队成员一般向领导者看齐，所以，领导者看重质量，大家就看重质量；领导者有欺瞒行为，大家就有欺瞒行为；领导者若对问题熟视无睹，这个团队就糟透了。

6.5.9　创新解决方案

Perl 设计者在 *Programming Perl* 书中提到，优秀的程序员有三大美德：懒惰、急躁和傲慢。为了减少总能量支出而不遗余力地努力的素质就是懒惰；受不了程序低效执行就是急躁；容不得对错误不管不顾就是傲慢。

要发扬程序员的美德，比如对于规则系统，如果不能构建上游的上千种场景，就可以思考新的解决办法。有人说做业务系统没什么追求和挑战，但是又有各种 Bug，小事不想做，大事做不来。其实“小事不小”。

6.6 踩过的坑和经验总结

下面讲讲笔者在工作中踩过的坑和经验总结。

6.6.1 简单的数字改动导致电商网站搜索功能失灵

众所周知，搜索功能是很多电商网站的重要流量入口，用户在网站上购买东西时，都会先通过搜索来查找自己想要购买的商品。所以，搜索功能往往是电商网站至关重要的功能。

某跨境电商网站曾出现一次搜索功能挂掉的情况。某天，该电商网站突然接到大量的用户反馈，称该电商网站的主搜功能无法使用，通过关键字搜索并没有结果展现，导致交易量急剧下跌。同时，监控告警提示该电商网站的交易量下跌、搜索展现下跌等。

该电商网站的开发人员第一时间排查问题，发现在几分钟前有一次代码发布，在快速回滚后功能恢复。他们通过代码审查，发现在本地发布中只有一行代码改动，即在数据的分页查询中，某开发人员将 pageSize 从 10 改成了 50。按理说这种小的改动不会影响到搜索功能，但一位有经验的开发人员第一时间指出了问题：由于某种原因，在代码中有一个默认设置：当搜索结果大于等于 50 的时候，就默认不返回任何结果。

导致该故障的因素有很多，下面进行简单分析。

◎ 这个默认设置当初是一个“临时方案”，但是一直没被改过。这就是技术债的问题。
◎ 如果不仔细阅读代码，就无法发现这个默认设置，因为这个故障在任意文档中都没有记载，并且没有被其他开发人员提到，更重要的是，在上线前未经过代码审查。这就是人、流程与文档之间的博弈问题。
◎ 在代码上线之前未经过测试。开发人员认为这只是一个很小的改动，不会有什么问题，所以就直接发布了。这就是质量意识淡薄的问题，也恰巧印证了墨菲定律：未经过测试的功能往往存在问题。

6.6.2 应用竟然被外部系统拖垮了

在众多线上故障中，有一类故障比较特殊，故障的起因可能来源于外部，也很可能是我们依赖的某个不重要的系统出现了异常，导致应用出现线上故障。这就是典型的“蝴蝶

效应”。

例如，在某个故障中，某网站的用户反馈网站响应变慢，甚至出现响应码为 500 的服务器异常错误，线上机器大量告警，出现频繁的 CMS GC。开发人员第一时间对机器的堆栈进行 Dump（转储），发现 ThreadLocal 中的一个对象占用了大量的内存。在分析代码时，他们发现在 ThreadLocal 中保存了一个 Map 作为本地缓存，在这个 Map 中保存了很多缓存对象，每次在读取缓存数据时，都会先从 ThreadLocal 中获取这个对象实例。在需要向缓存中保存数据时，就直接向这个 ThreadLocal 中的 Map 保存数据，导致这个 Map 越来越大。

这其实并不是导致频繁 CMS GC 的关键，因为在这个故障发生之前，线上环境平稳运行了很长一段时间。那么，到底是什么导致了频繁的 CMS GC 呢？

经过开发团队的辛苦排查，结果很是让人出乎意料：竟然是在应用中依赖的一个外部系统突然变慢，导致本应用的功能出现异常。由于外部系统响应变慢，请求外部系统的时间就会变长，所以在同一时刻存活的线程数变多了。由于每个线程都复制了一份 ThreadLocal 的数据，所以，ThreadLocal 占用的内存也变多了，这就发生了 CMS GC；又由于外部系统一直没有好转，线程数越来越多，所以发生了频繁的 CMS GC。

这就是典型的“蝴蝶效应”导致的故障：一个不重要的系统性能变差，触发了另一个系统发生 Bug，从而导致严重的线上故障。

6.6.3　HashMap 与并发

广大 Java 工程师对 HashMap 再熟悉不过了，HashMap 是非线程安全的，也是 Java 面试中的必考题目。但是，HashMap 在并发场景下出现的问题在很多公司和技术团队中都发生过，甚至在淘宝早期就发生过因为在并发场景下使用 HashMap 导致 CPU 被打满的线上故障。

阿里巴巴的研究员毕玄在自己的博客中曾经记录过这样一个案例：某天中午，开发人员收到大量的线上告警，提示某些机器的负载超过了 500。开发人员便赶紧登录线上机器，使用 top 命令查看了 CPU 的情况，发现 CPU 消耗接近 100%，于是又通过 shift h 命令查看线程的情况，发现并没有 CPU 消耗较高的线程，消耗量基本在 0.7%左右，但线程总数看起来很多。于是，开发人员通过 ps –eLf |grep java –c 命令统计了线程总数，发现一共有 700 多个线程，所以又连续执行了几个 jstack 命令查看线程堆栈，发现其中有 600 个线程处于 RUNNABLE 状态，并且都在执行 HashMap 的 get 方法。根据经验，基本可以断定问题是

由于在并发场景下错误使用 HashMap 导致的。

在该案例中，开发人员根据堆栈信息找到了相应的出问题的类名，然后找到了相应的 jar，用 jd-gui 反编译看了下相应的代码，发现在代码中使用此 HashMap 的方式如下：

```
public class xxxxx{
    private static Map<string,list> caches = new HashMap<string,list>();
        public void load(){
            caches = new HashMap<string,list>();
            // 向 caches 中放数据
            for…{
                caches.put(…);
            }
        }
    public List get(String key){
        return caches.get(key);
    }
}
```

堆栈中的信息显示有 600 个线程在执行 caches.get(key)。之前之所以在这个地方使用 HashMap，并且没有加锁，是认为 load 方法只会在 Spring 初始化当前这个 bean 的时候执行一次，但不幸的是随着业务的发展，这个 load 方法在数据变更时会被触发，可能出现向 Cache 里放东西的时候，而其他线程在取东西，这就直接导致了并发问题。关于 HashMap 的死循环问题，可以搜索相关资料了解细节。

当然，出现这个 CPU 消耗高的现象不是 HashMap 的问题，还是使用的问题，因此对这个问题的修复也很容易，就是用 ConcurrentHashMap 来替换 HashMap。

“high cpu usage in hashmap.get”是在 Google 上搜索“hashmap”时出现的热词，可见这个错误很常见，比如 Velocity、Hessian 都出现过类似的错误。Hessian 老版本中的类似 Bug 可以参见 http://bugs.caucho.com/view.php?id=1588；Velocity 老版本中的类似 Bug 可以参见 https://issues.apache.org/jira/browse/VELOCITY-718。

在以上案例中之所以出现故障，主要还是由于技术债导致的。由于在方案设计初期没有考虑到业务发展可能带来的变化，在无意中引入了“临时方案”，就欠下了技术债，进而导致故障的发生。

6.7　故障复盘流程及模板

在软件行业，一次线上故障的处理流程主要分为三个大方面：故障发现、故障处理及故障复盘，如图 6.18 所示。

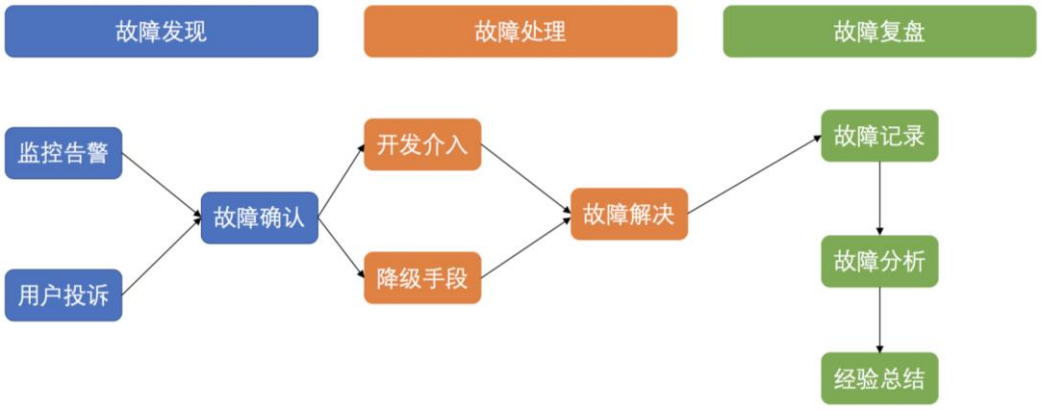


图 6.18

故障发现和故障处理是很多公司都比较重视的两个环节，故障复盘却被经常忽视，本节就来聊聊和故障复盘有关的事情。

6.7.1　什么是故障复盘

“复盘”原是围棋术语，本意是对弈者在下完一盘棋之后，重新在棋盘上把对弈过程摆一遍，看看哪些地方下得好，哪些地方下得不好，哪些地方可以有不同甚至更好的下法等。这种把对弈过程还原并且进行研讨、分析的过程，就是复盘。

“故障复盘”也是如上所述的过程，只不过不是针对棋局，而是针对一次“血淋淋”的线上故障。故障复盘会对故障处理的整个流程做一个“回放”，有利于我们深入分析和总结在故障处理的过程中哪里做得好，哪里做得不好，有哪些地方可以有更好的做法等。

6.7.2　为什么要做故障复盘

很多公司和团队都不重视故障复盘，究其原因主要有两点：对故障复盘有误解；不知道故障复盘的作用及重要性。

说到故障，就一定会有定责，不管在什么样的公司，这都是比较敏感的话题。发生一次线上故障，小则影响开发人员的绩效考核，大则可能导致高层引咎辞职，所以很多人在解决故障后都会下意识地抗拒重新提起发生故障的细节，尤其是和本次故障有直接关系的人或者团队。这其中的最主要原因是很多人对故障复盘存在误解，认为故障复盘的目的就是“分锅”和“甩锅”。不可否认，自我反思与相互批评是故障复盘中很重要的一个环节，但是，这并不是故障复盘的最终目的。故障复盘的最终目的其实是通过对已经产生的故障的反思，进行总结及优化。

笔者认为，故障复盘主要有如下目的。

◎ 对已发生的故障进行总结，避免再次出现同样和类似的问题。
◎ 找出团队的强弱项，优化人员分工和办事流程。
◎ 尝试找到更好的问题解决办法。
◎ 进行经验总结，分享沉淀，引起大家的广泛重视。

6.7.3 如何做故障复盘

前面介绍过故障复盘的目的和重要性，接下来看看如何进行一次有效的线上故障复盘，主要通过故障复盘的要求及流程进行展开。

1. 故障复盘的要求

故障复盘是故障处理中至关重要的环节，我们对其自然不能应付了事。一次规范的故障复盘需要满足如下 6 个要求。

◎ 故障复盘要及时，复盘时间距离故障解决的时间最好不要超过一周。
◎ 在故障复盘之前，相关人员应对本次故障的背景有基本了解。
◎ 在复盘过程中不批评、不表扬，就事论事，只陈述事实。
◎ 在故障复盘时要有完整的过程回溯，例如为什么会发生、第一时间如何解决等。
◎ 在故障复盘时对故障进行深层次的剖析，例如是否是流程存在问题、是否有更好的解决方案等。
◎ 在复盘后要有内容产出，例如故障复盘文档、复盘通知邮件、切实可行的行动（后续行动点）及行动点验收方案等。

阿里巴巴高级运维专家胡杨曾经总结过一套“RASA”理论，RASA 即 Review（回顾）、

Analyze（分析）、Summary（总结）和 Action（行动），如图 6.19 所示。该理论和以上介绍的故障复盘的要求不谋而合，都关注过程的回溯、深层次的剖析、经验总结及后续行动点的跟进。



图 6.19

2 故障复盘的流程

前面介绍了一次规范的故障复盘需要满足的几点要求，那么在一次完整的故障复盘过程中都需要做哪些事情呢？

故障复盘一般包含如下几个流程：故障记录、故障分析、复盘会议、行动点制定和行动点验收等，如图 6.20 所示。

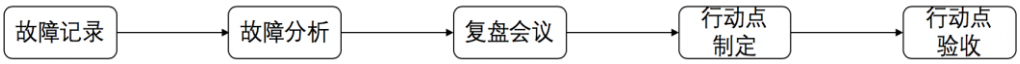


图 6.20

1）故障记录

故障记录一般由责任开发或者测试人员负责，有些公司可能由专门处理故障的服务支持人员负责。不管是谁来负责记录故障，都需要保证信息尽可能客观、准确，这样才能在后续的复盘过程中提供准确的信息。

故障记录主要包含如下内容：故障发生时间、故障描述、影响范围、主要模块、相关责任人、故障处理人员、故障处理时效、故障的当前状态、故障解决时间、故障解决方案、故障原因、故障定级等。

在故障记录过程中，有一个比较关键的步骤就是进行故障定级。故障定级根据不同的业务有着不同的指标，通常可以参考的指标有：影响用户数、故障恢复时间、用户投诉数、资金损失数、模块重要性等。

不同的公司对故障的级别有不同的定义，一般会有 P1、P2、P3、P4 等几类故障，这里的 P 是 Priority Level 的缩写。如表 6.1 所示是钉钉公开的开发文档中关于《应用故障定级标准》中故障等级的描述（参见 https://open-doc.dingtalk.com/microapp/operations/hn94hh）。

表 6.1

典型场景	业务模块	影响面	故障等级			
			P1	P2	P3	P4
使用应用（包括移动端、PC 端、管理后台）	核心模块	用户操作失败率>60%且每分钟失败量>100	30 分钟	1 小时	2 小时	3 小时
		10%≤用户操作失败率≤60%且每分钟失败量>100	1 小时	2 小时	3 小时	4 小时
		用户操作失败率>60%且每分钟失败量≤100	4 小时	12 小时		

可以看出，对于一个核心模块的功能，可根据用户操作失败率、每分钟失败量及影响时间等条件划分出 4 个等级。

有了故障定级，就可以针对不同级别的故障确定需要哪些人关注此次故障、故障规定处理时效，以及后续复盘会议参与人员及在故障定责时确定奖罚等级等。

2）故障分析

故障分析主要由故障相关责任人、故障处理人员等负责，旨在深入分析故障发生的根本原因、在故障解决过程中是否有可优化的步骤等。

在这个过程中，分析人员需要不断提出 5W1H 等相关问题，例如：

◎ 是什么原因导致了本次故障（Why）？
◎ 故障发生在什么业务、模块和功能点中（What）？
◎ 故障是在什么时间发生的、开发人员在什么时间开始接入的，以及在什么时候恢复的（When）？
◎ 故障发生在什么环境下、对哪些地区的用户有影响（Where）？

◎ 本次故障是谁发现的、谁解决的，以及谁该负主要责任（Who）？
◎ 故障是怎么定位的、怎么解决的，有没有更好的办法（How）？

通过不断追问，抽丝剥茧，洞察故障本质。

3）复盘会议

故障复盘会议通常由故障台工作人员或者故障责任团队牵头，邀请相关的开发负责人、质量负责人、产品负责人甚至运营负责人一起进行。在复盘会议上需要对本次故障及故障处理过程进行总结、讨论及反思。在某些情况下还会进行故障定性、故障定责等。

根据故障发生的模块、发生的原因等可以对故障进行定性和定责，即确定本次故障的性质、需要对本次故障负责的相关人员或者团队等。在定性和定责时需要明确出哪些个人（团队）对本次故障承担主要责任，哪些个人（团队）对本次故障承担次要责任，定责过程要公平、公正、公开，做到权责一致、边界清晰。

4）行动点制定

在故障复盘会议中，除了要对故障进行分析和总结，一件必不可少的事情就是执行后续行动点。根据具体的故障，行动点可能有多种，比如代码优化、流程优化甚至组织架构优化等。但无论具体的行动点是什么，制定的行动点都需要满足如下条件。

◎ 必须是明确的项，要有明确的行动点，不能是大而空的规划。
◎ 必须可落地，能够切实改善既有问题。
◎ 必须可验收，有明确的验收标准。
◎ 必须承诺验收时间，并且要在原则上按照承诺的时间完成。

5）行动点验收

在执行行动点之后，需要进行定期验收，还要对行动点周期性地跟进和确认。在验收时，要严格对照验收标准，确保行动点真实、可靠地落地。

6.7.4　故障复盘模板

前面介绍了什么是故障复盘、为什么做故障复盘，以及如何做故障复盘等内容，相信读者已经对故障复盘有了一些基本了解。接下来会给出一份故障复盘模板，如表 6.2 所示。

表 6.2

地　点		会议主题	
主持人		参会人员	
记录人		会议时间	

会 议 记 录

会议的主要内容如下。

一、故障描述

描述故障的发生时间、表现、主要模块、故障原因等。

二、故障影响面

描述影响用户数、故障恢复时间、用户投诉数、资金损失数等。

三、故障处理过程

（1）问题发现时间及发现方式。

（2）问题处理的详细过程。

（3）问题恢复时间。

四、故障原因

详细分析故障发生原因。

五、改进方案

新增监控告警项、监控报警时效性、离线对账等。

六、故障定级

故障等级		故障类别	

七、故障相关人员

报告人		处理人	
经办人		责任人	
责任团队		责任模块	

八、后续行动点

序　　号	行 动 点	验收时间	验收标准	负 责 人

故障复盘模板，其实就是在故障复盘会议上需要大家分析、讨论及总结的内容模板，也可以称之为故障复盘记录模板。

需要注意的是，故障复盘记录和前面介绍过的故障记录并不是一回事，故障记录指的是在故障发生之后对该故障跟进内容的相关记录，故障复盘记录则指的是复盘会议中的会议记录。

6.8　监控与告警

前面的章节介绍了很多关于线上故障的案例，在这些案例中，我们发现很多问题最初都是由一些监控及告警系统通知的。接下来就简单介绍监控与告警相关的知识。

6.8.1　监控的重要性

监控是每个公司都必不可少的重要风险防控手段。有效的监控可以帮助开发人员和运维人员快速发现线上的问题，从而保证应用服务的安全、可靠，以及业务的稳定运行。如果没有对线上系统进行监控，开发人员和运维人员便无法在第一时间主动发现系统发生的问题，只能被动等待用户的反馈，这在无形中延长了故障发生的时间。

有了系统监控，就可以化被动为主动，在用户反馈前发现并解决问题，提升用户体验，减少用户投诉。

6.8.2　监控哪些内容

前面讲了监控的重要性，那么，一个完善的监控系统应该监控哪些内容呢？开发人员和运维人员需要重点关注其中的哪些监控指标呢？以一个分布式网站为例，对该网站的监控一般包含系统监控、应用监控、进程监控、服务监控、业务监控、调用链监控等。下面列举在各项监控内容中需要我们重点关注的指标。

1. 系统监控

目前有很多应用都部署在 Linux 系统上，对于线上服务器的系统监控是一个比较重要的指标。我们通常关心的服务器相关指标有 load、cpu、mem、traffic 等。大多数情况下，在线上出现问题时，往往伴随着某些指标的异常。而且，在问题发生之前，某些指标会提前显示异常。

对于 Linux 机器，我们可以直接使用 Zabbix 进行系统指标监控。Zabbix 提供了系统

监控模板 Zabbix Agent，如图 6.21 所示。



图 6.21

Zabbix 是由 Alexei Vladishev 开发的一种网络监控管理系统，基于 Server-Client 架构，可用于监控各种网络服务、服务器和网络机器等状态。在客户端如 UNIX、Windows 中安装 Zabbix Agent 之后，可以监控 CPU 负荷、网络使用状况、硬盘容量等状态。

2. 应用监控

除了系统级别的监控指标，最重要的就是在服务器上部署的应用的监控指标了，例如 JVM、Nginx、MySQL 等相关监控。

在采用 Java 开发的系统中，对 JVM 的监控是至关重要的，其中比较重要的几个指标有 JVM 的内存使用情况、GC 次数、GC 耗时、JVM 线程数等。Java 平台为应用程序、设备和系统等植入了有管理功能的 JMX 框架，JMX 框架提供了关于 JVM 的大部分核心信息，应用系统可以通过接入 JMX 框架来实时读取 JVM 的 CPU、内存等信息。前面介绍过的 Zabbix 也提供了 Zabbix JMX 框架，支持通过该框架实现对 JVM 的监控。

除了 JVM，应用的正常运行还依赖 Nginx、MySQL 等基础设施，对这些基础设施的监控同样重要。对于 Nginx，我们一般关注其接收请求数、处理请求数等；对于 MySQL，我们一般关注 SQL 语句的执行时间、每秒请求数和慢 SQL 等。

3. 进程监控

进程监控，其实可以归类为系统监控或者应用监控，其监控指标也较为简单，就是进程有没有活着。但是，进程监控比较重要，值得在这里单独介绍。例如，对于 Java 这种单进程应用来说，进程挂了，就意味着整个系统都挂了，特别是对于某些定时任务应用，我们要对其进行活检监控，保证进程正常执行且没有挂掉。

4. 服务监控

随着近几年微服务等技术的不断流行，很多应用都是分布式部署的，各个应用之间是通过 RPC 或者 HTTP 服务来通信的。所以，对于服务的监控也越来越重要。

服务监控的指标一般包括请求数、QPS、RT、错误数等，可以通过 Web 服务器的 access log 或者应用的日志获取这些指标。

5. 业务监控

无论我们对线上应用做多少监控，最终目的都是保证业务的正常运行。所以，对业务指标自身的监控也是十分重要的。

业务监控一般没有统一的方法，对于不同的业务，需要监控的指标不尽相同，监控方法也千差万别。比如，对于一个电商网站来说，其业务指标可能有下单人数、下单成功数等。

6. 调用链监控

现在的互联网服务通常是由复杂的大规模分布式集群实现的，互联网应用构建在不同的软件模块集上，这些软件模块可能是由不同的团队使用不同的编程语言实现的，也可能部署在几千台服务器上且横跨多个不同的数据中心。因此，我们需要一些工具来帮助理解系统的行为和分析性能问题。

调用链监控在微服务系统中比较常见，一般用于跟踪一个请求从发起到返回的全过程，方便查看一个请求在各个相关系统中的调用情况。调用链监控是跨系统的监控，需要在请求发起时分配一个可以唯一识别本次调用请求的 traceId，这个 traceId 在整个调用链中一路透传下去，之后在每个系统的调用日志中都输出该 traceId。在所有日志都汇总起来后，便可以根据日志分析本次调用的流程。

目前，业内的调用链监控大多是参考 Google 论文 *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure* 实现的。

6.8.3 告警

监控与告警是一对分不开的兄弟，其实，我们做监控的目的是发现问题，然后通过告警方式将问题通知给开发人员及运维人员，让其快速定位并解决问题。告警信息一般通过邮件、短信、电话、钉钉等方式发送。

告警信息需要包含时间、机器 IP、告警原因、告警描述和错误日志等重要信息，以便于开发人员快速排查问题，如图 6.22 所示。



图 6.22

6.8.4 监控的架构

要想构建一个智能监控告警系统，就需要重点关注监控和告警，其中涉及的指标就是系统监控、应用监控、进程监控、服务监控、业务监控和调用链监控等。一个智能运维监控系统，在架构设计上从低到高可以分为 3 大模块，总计 6 层，如图 6.23 所示。

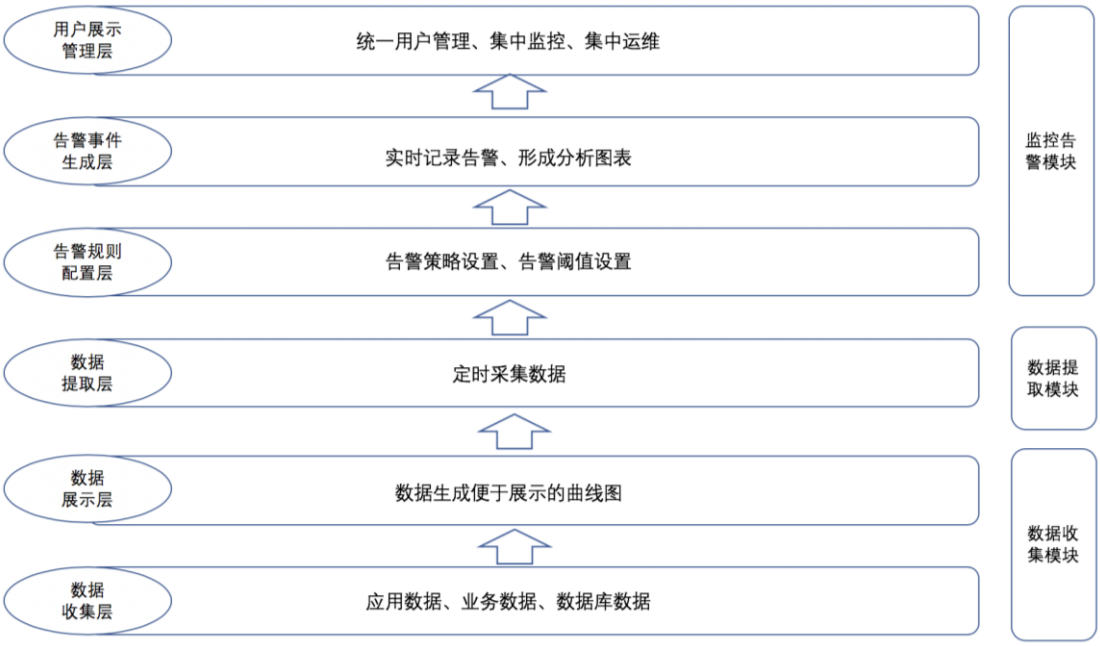


图 6.23

下面对如图 6.23 所示的各层进行详细讲解。一个完整的监控告警系统的整体架构一般分为三个重要模块：数据收集模块、数据提取模块及监控告警模块。

数据收集模块主要负责数据的收集及展示，主要包含数据收集层和数据展示层。

◎ 数据收集层位于整个监控架构的底层，主要收集应用、业务及数据库相关的元数据。

◎ 数据展示层位于第 2 层，主要将数据收集层采集到的数据进行可视化展示，一般将其绘制成折线图、柱状图等，方便运维及开发人员查看数据的趋势及进行对比。

数据提取模块主要负责数据的提取，衔接数据收集模块和监控告警模块，主要包含数据提取层。数据提取层位于第 3 层，主要负责对数据收集层采集到的数据进行二次加工，以及规范化和过滤处理，并将需要的数据向上传递到告警模块。

监控告警模块主要负责监控告警的配置及送达，包含告警规则配置层、告警事件生成层及用户展示管理层。

◎ 告警规则配置层位于第 4 层，主要承接来自数据提取层的数据，并根据这些数据进行告警规则、阈值、告警联系人及告警方式的配置。
◎ 告警事件生成层位于第 5 层，根据告警规则配置及监控数据生成需要告警的事件，并且负责将告警事件通知给相关联系人。
◎ 用户展示管理层位于顶层，是一个 Web 展示界面，负责将监控统计报表、告警详情等信息统一展示给用户。

6.9 应急处置

在日常工作中，线上环境出现意外情况是无法避免的，应急处理就是在故障发生时快速恢复服务，避免或减少故障带来的损失，以及对用户的影响。

应急处理原则如下。

◎ 应该第一时间恢复系统，而不是等彻底解决问题后恢复系统，要快速止损。
◎ 在明显造成资金损失时，要第一时间升级、快速止损。
◎ 在快速上线时一定要慎重，防止造成新的故障。
◎ 定位故障，快速启动应急流程与止损方案。
◎ 当运维负责人不能在短时间内解决问题时，必须进行升级处理。
◎ 尽量在不影响用户体验的前提下处理问题，保留现场。

线上应急处理一般为 6 个阶段，如图 6.24 所示。要记住应急只有一个目标：尽快恢复系统、消除影响。不管处于哪个阶段，首先都要想到恢复系统，恢复系统不一定能定位问题，也不一定有完美的解决方案，但有利于保留现场、定位问题、处理故障和回顾故障。

应急处理		
	外部流程	内部流程
上报故障	将线上故障通过群、在线客服或电话反馈给客服	将已知。疑似故障或数据问题通过邮件或群的方式反馈提交给 IT 部门排查
接收故障	客服录入工单系统：提供详细的问题描述： 1．用户、设备等基础信息 2．业务参数（如订单号、申请号） 3．场景步骤描述 4．影响范围（个案、批量）	1．内部业务方反馈问题表征 2．内部业务期望的修复时间、紧急程度等 3．NOC 对该问题的初步判断
定位故障	1．问题分类 2．预计修复的时间	1．问题分类 2．预计修复的时间
处理故障	1．永久修复 OR 暂时修复 2．IT 部门内部排期优化到生产上线	1．永久修复 OR 暂时修复 2．IT 部门内部排期优化到生产上线
反馈通知	1．客服自行处理或者收到工单反馈 2．通知报障人	1．NOC 通知内部业务方定位结果 2．NOC 通知内部业务方修复结果 3．NOC 通知测试团队生产 BUG 情况
回顾故障	1．记录故障处理全过程 2．故障整改计划 3．升级故障问题库	1．记录故障处理全过程 2．故障整改计划 3．升级故障问题库

图 6.24

下面对应急流程进行详细讲解。

1. 定位故障

在定位故障时，我们可以通过系统最近发生的变化，以及系统层、应用层和中间件层监控来定位故障，例如：

◎ 系统最近是否上过线？
◎ 依赖的基础平台是否升过级？
◎ 依赖的系统是否上过线？
◎ 是否有线上营销活动？
◎ 网络是否稳定？
◎ 业务量是否有变化？

对系统层一般监控如下指标。

◎ 系统的 CPU 使用率。
◎ 平均负载（Load Average）。
◎ 内存（Memory）。
◎ 网络与磁盘 I/O。
◎ SWAP 使用情况。
◎ 线程数。
◎ 文件描述符（File Description）等。

对应用层一般监控如下指标。

◎ 接口的响应时间。
◎ 每秒查询率（QPS）。
◎ 调用频次。
◎ 接口成功率。

对中间件层一般监控如下指标。

◎ 数据库的负载、慢查询、连接数等。
◎ 缓存的连接数、占用内存、吞吐量、响应时间等。
◎ 消息队列的响应时间、吞吐量、负载、堆积情况等。

2. 处理故障

处理故障要以定位故障为基础，必须清晰定位故障产生的根本原因，在提出解决故障

的有效方案之前，不要用不确定的方法来尝试修复故障，这样可能导致引入新的故障。

3. 回顾故障

在处理故障后，应急团队与相关方需要回顾事故产生的原因及应急过程的合理性，并提出整改措施，主要聚焦于如下几个问题。

◎ 类似的问题还有哪些没有发生？
◎ 做了哪些事情，故障就不会再发生？
◎ 做了哪些事情，即使发生故障，也不会有负面影响？

3

第 3 篇

管理探秘

第 7 章

为成为技术主管做准备

笔者在这个行业里面耕耘 10 年有余，在跟年轻朋友交流的时候，很多朋友都会问笔者一个问题：“我用什么方法才能成为像我主管那样的人？”

这个问题回答起来有点尴尬，因为很难用只言片语说清楚应该用什么样的既定方法去实现。“主管”这个人设被赋予了大量的标签和职能，其修炼并非一朝一夕即可完成，与其自身的思考深度及机遇也有很大关系。笔者思考良久，决定记录并分享自己的成长，希望对朋友们有所帮助。

7.1 构建自我阶段性目标

笔者读过 TED 大会新锐演讲者简·麦戈尼格尔写的《游戏改变世界》这本书，其中写到：满意的工作从两件事开始，一是有明确的目标，二是有实现这一目标的可操作性步骤。

有明确的目标可激励我们采取行动，让我们知道自己该做什么；有可操作性步骤可确保我们立刻朝着目标前进。我们需要对自己在做的事情有一张明确的阶段性画像，这张画像一定是自己内心所向往的。

或许你是一个虚无主义者，认为自己没有什么目标，而制定目标这件事跟自己也没有什么关系，但这并不影响你去寻找自己的目标，这跟不认同万有引力定律但是出门可能摔倒是一个道理。

作为程序员，我们生活在一个快节奏的时代：为了团队的业绩夜以继日地加班、敲代码，没有时间谈恋爱、思考甚至吃饭，更别提花时间规划自己的目标。再加上，在斑斓多彩的社会环境下，我们需要大量的物质去追求幸福感，很容易迷失其中，忘记自己的根本

需求，把挣钱作为唯一目标。

当然，挣钱这个目标本身没有任何问题，但是它和你的其他目标一样，只是一个阶段性的目标。有这样一个故事：有一位武术大师想学习李小龙的所有武功，李小龙拿出两杯倒满水的杯子，告诉这位武术大师，如果想把第 2 个杯子里的水倒入第 1 个杯子，就需要先把第 1 个杯子里的水倒掉。所以，如果想找到自己的真实目标，就需要清空自己的错误思维，不要把当前目标当成唯一目标。

既然因为爱好或生活选择了程序员这个职业，我们就更应该清晰地构思自己的下一个目标，让自己知道程序员的进阶方向在哪里。在忙碌的一周里找一个完全属于自己的下午，哪怕只有一个小时，不接听电话、不看微信，也不看技术文档，尝试在一张白纸上写下自己的最终目标，然后构建自我的阶段性目标。

如果大家不知道该如何下手，则不妨参考 Steve Pavlina 的 *How to Discover Your Life Purpose in About 20 Minutes* 文章来尝试在短时间内找到并构建自己的目标，这里给出如图 7.1 所示的思维导图，以总结 Steve Pavlina 在这篇文章中介绍的快速构建自己的目标的步骤。

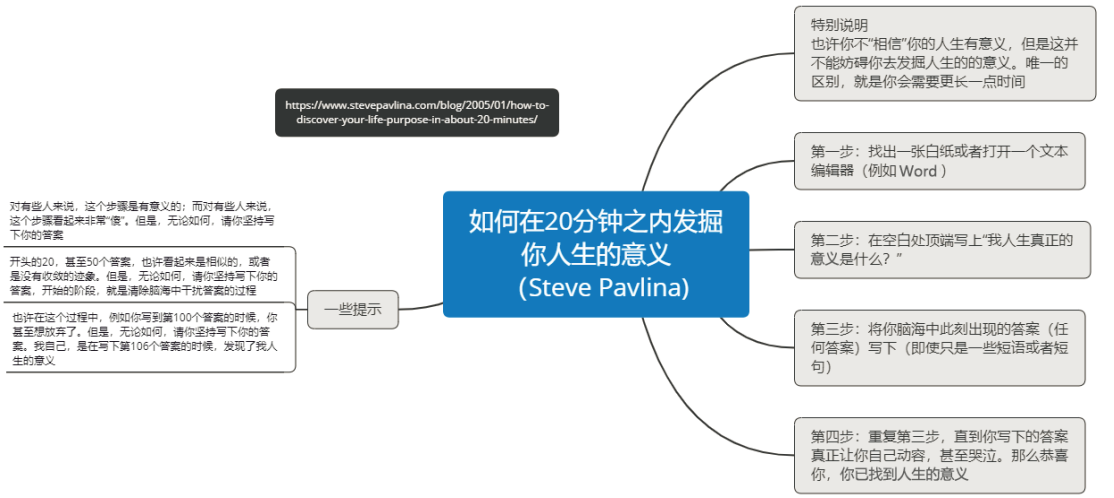


图 7.1

7.2 体验自己的目标身份

拥有一个清晰的目标，便成功了一半。如果已经构建了技术主管这个阶段性目标，接下来就需要反复做唯一的选择，每天从起床那一刻开始，就应该体验技术主管的身份。那么，具体需要做些什么呢？

1. 必备技能准备

业务人员往往会犯这样的错误：只关注业务实现，对算法和组件只停留在“用”的阶段，不会深入了解算法、网络传输协议和组件实现思想等更进阶的内容。我们可以根据在目前工作中所用到的技术体系去深入学习，这样才会具备技术主管的主要技能：追求最优化的实现，剖析问题的本质及整体业务架构。

2. 从技术主管的视角去工作

这里讲的“视角”是一种意识形态。在业务上，技术主管的角色一定不能有业务边界，需要跳出自己的业务范畴，以上下游的高度去思考，把自己的业务和其他业务串联成整体看问题；带着业务目标去看待每一次的项目实现，充分考虑稳定性、可扩展性和每一次的项目实现对目标的影响。假设一个方案在实现后刚好能满足用户的需求，那么从技术主管的视角来看，我们应该思考是不是可以超越这个“刚好”，把业务做得更完美。

3. 学会用户思维

在设计、研发产品的过程中，我们往往会带入自己的专业信息及思维方式，使得用户在接触我们的产品时需要花大量的时间去学习和实践，最让人崩溃的产品大致如此。技术主管不仅需要具备专业知识，还需要把这些专业知识传递给用户，在发现产品设计中存在这样的问题时也应该及时纠正。

4. 建立经济视角

作为技术主管，建立经济视角非常重要，因为技术主管应该对团队的 ROI 有更周密的考虑。研发部门往往被视为成本中心，技术主管却是商业战场上的特种兵，在其领导下的每一个人及写出的每一行代码都可能对公司的业务有极大的影响。无数事实证明，一个有超强执行力和自我思考能力的研发团队可以改变整个战局。所以，作为战术领袖，技术主

管一定要有经济视角，充分掌握实现更优的投入产出比的方法，合理思考事情怎么做、谁来做及会怎样，以实现公司的商业价值为终极目标，化解研发团队是成本中心的误解，将研发团队塑造成效率中心和成果中心。

5. 成为内部 BD（商务合作）

这听起来可能有点令人匪夷所思，但正如上面所讲的经济视角一样，项目以实现公司的商业价值为目标，当以结果为导向去工作的时候，我们就会发现自己需要做的不再局限于写代码，而是要对资源做整合，同时需要通过沟通来协调不同部门的资源并且建立集体意志，让所有人都认同并围绕相同的目标去工作，甚至把不同部门的人临时编排在一个虚拟组里，而这一切的促成都离不开 BD。

6. 规划团队从规划自己开始

具体做法是合理安排时间和做好工作计划，建立阶段性目标并实现，用阶段性的成果去建立认同性的动机，然后提高标准来完成进阶，用每一次的成功去激励自己坚定地做唯一选择。

7. 数据驱动

指利用数据的力量去激活自己和团队的状态，关注结果数据且分享给团队，并对数据做出实时反馈，从成功的数据中得到鼓励，从失败的数据中总结出不足并调整方案。但不管是成功还是失败，都是已经发生的，重点还是现在，所以请坚定对未来的信心并且把希望传递给自己的团队，带着希望去告诉大家失败数据产生的原因及接下来的做法。

7.3　勇于抓住机会

如果你一直处于“有经验的技术主管”状态，那么你需要做的就是保持对事件的敏锐洞察能力，当一件能够让你独立完成且把能力付诸实践的任务发生时，请对自己说：“这就是我的!”在接到这样的任务以后，你需要快速地从对过去的总结和对未来的焦虑状态中抽离出来，平静地用自己的经验去分析业务目标和实施路径，所有这一切都应该是那么自然和熟悉，因为你时刻为此准备着，而且这一切本就属于你。

第 8 章

从自我管理转为管理团队

敏捷大师琳达·瑞思（Linda Rising）是变革模式（Fearless Change Pattern）的提出者，她曾说过：“我提出的这些变革模式的确看起来非常直白，但是直白绝不意味着简单。我努力在做的，就是让大家不但能够理解这些模式，还能够应用这些模式。”

程序员需要突破技术桎梏，在带人和带团队上不断成长，这似乎看起来也非常直白，却做起来绝对不简单。

俗话说，学而优则仕。在如雨后春笋般出现的各互联网和软件公司中，有越来越多的技术达人因为在技术上的优异表现，晋升为技术团队的管理者和领导者，这也被人们戏称为“编而优则仕”。然而，对于这些技术达人来说，从单枪匹马的个人贡献者突然被提拔为一呼百应的技术团队领导者时，随之而来的，除了兴奋，还有各种困惑和不适“症状”。因为，对于技术专家而言，在大多情况下做好自我管理就可以取得成功；而对于团队管理者而言，把一个队伍带好，注定是超越自我管理的挑战。

接下来就从对什么是领导力的追问开始，一步步探究技术团队管理者从自我管理到带好团队这一转变的背后，到底需要怎样的能力支撑。

8.1 什么是领导力

什么是领导力？要回答这个问题，必须解答什么是管理，以及管理和领导力的区别。

管理学教科书通常将管理定义为：管理是指在特定环境条件下，对组织所拥有的人力、物力、财力、信息等资源进行有效决策、计划、组织、领导、控制，以期高效地达到既定的组织目标的过程。简单概括一下：管理的核心，就是利用组织的资源把事干好，从而达到组织的目标。

如果需要用一句话对领导力进行定义，那么管理大师彼得·德鲁克的这个定义，一定是不二之选：领导力就是把一个人的视野提到更高的境界，把一个人的成就提到更高的标准，锤炼其人格，使之超越通常的局限。

如果对比上述对管理和领导力的定义，我们不难发现一个核心区别：管理的核心在于事，领导力的核心在于人。在彼得·德鲁克对领导力的定义中更多地关注人：从一个人的视野到一个人的成就和人格。

进一步说，领导者的视野超越了一般管理者所关注的具体工作或者 KPI，转而关注最核心的人，通过把所领导的人的视野提到更高的境界，来开发一个人的潜力和持续的创新动力，让他获得自己以前想都不敢想的成就。这也和著名的管理学大师汤姆·彼得斯（Tom Peters）的观点一脉相承：领导是做不一样的事，它不是让你做得卓越，而是激发别人表现卓越。

同时，领导者的一切行为都受其价值观的支配，好的领导者是言行一致的人，其所表现的对人的关注，并非由于其所处的位置，而是其价值观和信念使然。当一个领导者发自真心地表现出对人的重视，对把他人提升到更高的境界、获得更大的成就表现出浓厚的兴趣时，这个领导者一定是有人跟随的，并且是被心甘情愿跟随的，这样的领导者和其带领的团队所迸发的力量也是巨大的。

8.2　如何构建领导力

基于笔者近年来从技术转管理的培训经验，如果给国内大多数从技术转管理的团队管理者做一个能力储备的“X 光”检查，则得到的结果往往是这样的。

（1）领导力方法论：大部分人都听过一些领导力或者管理方面的“大道理”，但是对领导力的规律和需要秉持的原则缺乏系统梳理；缺乏对领导力和管理方法论的系统反思和总结，普遍处于“能力真空”状态。

（2）提升领导力的具体工具和方法：通过观察、模仿和学习，有一些基本领导和管理手段，但是对这些管理手段、方法和方式的适用场景和效用缺乏反思和总结。同时，往往把注意力集中管理工具上，如果缺乏合适的方法论支撑，通过这些看起来不错的管理工具并不能达到预期的效果，甚至可能南辕北辙。

这里给出一套可结构化地提升技术领导力的框架，叫作“技术领导力 4D 框架”，如图

8.1 所示。在这个框架中，技术领导力被分为了 4 个维度，涵盖了技术团队管理者的绝大部分工作。



图 8.1

下面对技术领导力 4D 框架中的 4 个维度分别进行解析。

8.2.1 维度 1：提供清晰的领导力风格，并以信任感作为基石

兰西奥尼在《团队协作的五大障碍》中指出：“企业最根本的竞争优势既不来自资本实力、发展战略，也不来自技术，而是来自团队，因为一个优秀的团队是非常强大而且极其难得的。”然而，阻碍团队成员齐心协力、目标一致、绩效优异的最主要因素，就是缺乏信任感。

这里所说的信任感，指的是在团队成员彼此之间、管理者和团队成员之间，都能够充分相信对方，并敢于暴露自己的不足，无须在团队协作中有太多自我保护，每个成员都相信其他成员的所有行为或者反馈出于好意。只有建立起了这种信任感，大家才可能畅所欲言、不惧冲突，在开放的讨论中达成一致，并敢于创新和尝试，从而实现团队的高绩效。这种基于暴露弱点的信任感，正是一位技术团队领导者需要和团队建立的重要关系。

除了这份信任感，团队管理者还需要提供清晰的领导力，才能为团队协作达到高绩效提供坚实的基础。而清晰的领导力往往是技术团队管理者忽视的领域，也是需要提升的领域。有这样一个案例，管理者在向员工布置任务时，会把任务“重复”5 遍：

管理者：我有一个任务要交给你，任务是这样的……（**第 1 遍进行详细的任务说明**）

员工：好的，我知道了。

管理者：稍等，你能重复一遍这个任务吗？

员工：好的，您交给我的任务是这样的……（**第 2 遍重复任务是什么**）

管理者：你有想过我为什么会布置这个任务吗？

员工：我理解，您布置这个任务的原因是……（**第 3 遍谈论任务的目的**）

管理者：在这个任务的进行过程中，你觉得会出现什么意外情况？遇到什么情况你会向我汇报？遇到什么情况你能自己决定？

员工：我理解，在这个任务中，可能会遇到的意外情况有……（**第 4 遍明确风险和汇报标准**）

管理者：如果这个任务让你自己做，你会有什么更好的建议和想法吗？

员工：如果这个任务让我自己做，我会……（**第 5 遍让员工发挥主观能动性**）

与这位管理者将任务“重复”5 遍相比，大多数管理者常常希望通过一个眼神就能让员工明白自己的意思，或经常把“我不希望再说第 2 遍”挂在嘴边，这往往是因为对团队管理者和团队成员之间的默契程度预估过高，所以其结果往往并不如意，很多时候需要重新返工或补救，实际上成本更高。

8.2.2 维度 2：了解业务，并带领团队达到高绩效

对于很多从技术专家到团队管理的管理者来说，似乎只需要懂技术就可以了，业务都是销售人员才应该去关心的问题。真的是这样吗？实际上，我们可以从软件开发的敏捷方法论中去探询这个问题的答案。

在敏捷宣言的四条敏捷核心价值观中，其中非常重要的一条就是：能工作的软件优于详尽的文档。

实际上，对于一个软件团队来说，产生价值的最主要途径（如果不是唯一途径的话）就是交付能满足用户需求的可以工作的软件，所以需要强调与用户合作及响应变化，也需要在团队层面通过重视个体和互动来激发团队成员，从而实现高绩效。

因此可以认为，对技术团队管理者来说，了解组织的业务，了解软件如何为组织创造价值（例如营收），才能更好地确定战略方向、制订计划及实施，进而交付对用户来说最有价值的软件。

从另一个角度来说，在技术管理者带领团队确定战略方向，制定技术和组织实施的过程中，技术管理者本身也在进行角色从“演员”到“导演”的转换和突破。在没有做技术管理者之前，技术专家在遇到问题时往往都冲在最前面，就像舞台聚光灯下的演员。管理者则需要对整台戏进行策划，寻找最合适的演员，然后给每个演员都分配明确的角色，让演员们在舞台上都有最出色的表现，自己却可能不上台。因此，从技术到管理的一个非常重要的突破就是从台前的“演员”变为幕后的“导演”。因此，“了解业务，带领团队高绩效”其实也是管理者本身升级和突破的过程。

8.2.3 维度 3：发展自己和团队成员

美国商业领袖、通用电气（GE）前 CEO 杰克·韦尔奇曾经说过，“成为领袖之前，成功就是让自己成长。成为领袖之后，成功是让别人成长”。

为什么发展自己和团队成员那么重要？前面分析过管理和领导力的区别，我们知道，管理更多地关注事，因此比较看重执行力；而领导者知道员工在一个组织中最大的诉求是成长，因此会花很多时间帮助团队成员提升能力，获得成长，将工作做得更加高效和高质量，从而获得组织和个人的双赢。

著名的心理学家、行为科学家、美国犹他州大学管理学院教授弗雷德里克·赫茨伯格

（Frederick Herzberg），在进行大量的调查之后总结了最能够激励员工的因素，排在前三位的分别是：成就感、认可和工作本身。

而发展自己和团队成员，就是为了更好地让团队成员获得成就感，进而实现团队的高绩效。这实际上也正是“从技术到管理”需要完成的“关注点从事到人”突破的一部分。原因非常简单，技术团队通常“编而优则仕”，即优秀的程序员或者技术专家通常被提拔为领导者。虽然人们认为聪明的技术专家也一定会是优秀的领导者，但事实证明，在很多情况下这只是个美好的愿望而已。技术专家面对的问题域主要是“事”层面的技术、系统、算法和程序等，领导者面对的问题域则主要是“人”（人的情绪、观念、想法等），这正是引发从技术到管理这一挑战的源头，也是技术团队管理者需要突破的一点。

对于刚从技术专家转换角色的领导者，首先需要实现的突破就是忍住“我自己做”的冲动，转为发掘和培养一个最合适的人，然后选择最适合他的工作，把事做到最好。

8.2.4　维度 4：塑造未来

在领导者对团队所产生的影响中，愿景所带来的改变是最深远的。与此同时，有效能的领导者往往是有愿景的，这样的领导者通常被称为“愿景型领导”。

愿景型领导通常会设立高远的抱负或目标，从而形成企业强大的追求拉动力，使各级管理者沿着充满野心的甚至“胆大妄为”的理想不断前进。它基于这样一种哲学假设：人的生命是短暂而有限的，职业者应将其个人发展融入企业的发展中来共同实现社会价值。因此，人的潜能是无限的，并且应该以高不可攀的目标将其激发出来。

马云实际上就是愿景型领导的例子。马云在创业伊始，便通过自身的人格魅力凝聚了团体，并将自己的梦想作为一种使命传播到团队中，使得每个人都有一种使命感，然后提出了三个目标：

（1）建立一家可生存 102 年的公司；

（2）建立一家为中国中小企业服务的电子商务公司；

（3）建成世界上最大的电子商务公司，进入全球网站排名前十。

愿景更多指的是一个人的梦想，是着眼未来、富于激情的。马云之所以能成功，是因为他还能把这个愿景立足于现实。在公司发展过程中，不断有新人加入，那些认同马云的愿景并且留下的人，都被马云基于这些愿景而塑造的未来所激励，并一起建立了今天进入

世界五百强的阿里巴巴，同时为这些愿景的实现打下了坚实的基础。

因此，对一个技术领导者而言，领导力中非常重要的一环就是：和团队一起确立共同的愿景，带领追随者实现梦想，同时协助他人实现自我和超越自我。

8.3 让自己成为 T 型人才

IDEO 是一家全球创新设计公司。多年以来，人们都把 IDEO 作为学习目标，不停地研究为什么其设计师能源源不断地产出创意十足的设计方案。IDEO 的首席创意官 Paul Bennett 在谈他们的人才选择策略时给出了答案：IDEO 在进行人才选择时不仅仅选择拥有工业设计、交互设计、沟通设计或商业设计专业背景的人才，也会选择广泛涉足宏观经济学、认知科学、食品科学、人类学、应用语言学和基因工程等领域的人才，IDEO 将这种人才称为“T 型人才”。

我们也可以将 T 型人才简单理解为“一专多能”，T 字的横轴表示跨专业领域的思考能力，T 字的纵轴表示精深的专业能力。我们之所需要这样的人才，是因为他们拥有广泛的背景知识，更利于沟通，往往对解决方案持开放态度，更容易倾听和接受对方的建议，而且可以在团队协作中从不同的角度提出创新点，给予团队新的视角。

T 型人才作为项目推进者，除了能给团队带来专业的指导，也能从不同的方向引领团队完成任务，所以 T 型人才通常具备如下能力。

◎ 有专业的深度，也有思维的广度，能够跨界思考和探索。
◎ 专注于当下，且对外部世界持开放态度，能接纳不同的视角和观点。
◎ 关注问题的根源，不急于设计解决方案，而是从系统的角度整合方案设计。

在互联网的发展过程中同样需要大量的 T 型人才加入。比如，在一个 AI 项目中，我们往往会选择一些懂算法的应用工程师、会编写代码的算法工程师和能够很好地理解算法的产品经理，因为这样的人组合在一起才能更高效地沟通方案，以及接纳彼此，通过长期的思维碰撞还会有更多的新颖创意产生。同样的事情在 IDEO 也发生着：IDEO 为了解决一个“减少当前家用水的消耗”的问题，将 1 个精通水力学的工程师、1 个负责揭示人们在家用水潜在需求的人因调研专家，还有多个完全没有任何相关专业背景的设计师组合起来，这些专家帮助讲述精彩的故事，使客户能够看见和感受较为抽象的事物。因此，如果某团队的成员都有着专业的背景及广泛的领域背景知识，那么这个团队一定是极具创造力的。

所以，笔者目前所在的公司在面试主管及以上级别的人才时，不仅仅考核候选人的专业知识，也会从他过往的经历中寻找跨专业思考的影子。非常典型的一个例子：我们在面试一名项目负责人候选人时发现，面对如何保障团队按时按量交付项目的问题，该候选人不仅完美讲解了他专业的项目管理和风险管理经验，还讲解了他是如何通过积极心理学的知识去分析和保持团队成员的幸福感和持续激情的；最后，我们录用了这名候选人，并把他加入我们更多、更重要的项目推进中。事实证明，该候选人精深的专业能力和跨专业思维能力在很大程度上帮助了团队快速形成凝聚力和前进的动力。

为了让自己成为 T 型人才，我们需要做充分的准备。

◎ 离开舒适区，学会挑战自己，让自己所掌握的技能变得更为精深；与同行保持交流，从中掌握不同的设计方案和不同的思考方式。
◎ 坚持阅读及与不同领域的人交流，以接纳更多的信息和观点。
◎ 创造需求，及时应用和实践自己掌握的知识，这也是让知识变得更有价值的好方法，而不仅仅是让知识成为一种可阅读的信息流。
◎ 尝试训练自己从多个角度分析问题，或用不同的思维思考问题。
◎ 给学习和应用设定目标和里程碑，让自己一步步地成为 T 型人才。

8.4　高效时间管理

博恩·崔西（Brian Tracy）是美国首屈一指的个人成长权威人士，获得 C.P.A.E.美国演讲家最高荣誉，还为 500 多家公司提供咨询服务，包括 IBM、美国麦道公司等。然而，博恩·崔西获得巨大成功的秘诀简单得让人出乎意料：找出成功人士做事的方法，然后加以效仿，你就能取得同样的成就。

博恩·崔西在校求学期间学习成绩很差，还没有毕业就辍学了。为了生计，他只能挨家挨户到处奔波、推销产品。有一天，他质问自己：“为什么他们比我做得更出色？”然后开始采取一种彻底改变了自己的生活和事业的行动：拜访成功的金牌销售人员，向他们请教成功的秘诀。那些金牌销售人员都不吝赐教，博恩·崔西如法炮制，销售额果然直线上升。由于销售业绩出众，他被提拔为管理者。之后，博恩·崔西采取了同样的行动，只不过这次拜访的对象是那些成功的销售主管，博恩·崔西同样把他们的成功经验充分运用到自己的工作实践中。

时间管理对于任意一个人来说可能都是或大或小的困扰，因为时间对于每个人而言都

是公平的——无论贫穷还是富有，每个人一天都只有 24 个小时。而对于技术团队的管理者而言，时间管理显得尤为重要，因为技术团队的管理者和团队成员大多数是脑力劳动者，对于脑力劳动者特别是优秀的脑力劳动者而言，单位时间的产出价值是非常可观的。因此，如果能够做好时间管理，技术团队增加的效能将是巨大的。

如果说本书的其他章节用于帮助团队管理者把自己和团队成员培养成为优秀的员工，从而提升其所创造的价值，那么本节的内容就是通过高效时间管理，让优秀的员工可以有更多的时间或者更高的效率创造价值，从而实现组织和自身的双赢。

那么，如何才能做到高效时间管理呢？其实博恩·崔西已经给出了答案：只需效仿优秀、成功管理者的高效时间管理方法，就可以获得很好的效果！

因此，本书深入学习了管理大师彼得·德鲁克、亚历克·麦肯齐、戴维·艾伦、斯蒂芬·科维等的相关著作，也与技术管理社区的朋友们进行了广泛交流和探讨，总结了技术团队管理者的高效时间管理“招式”，这些招式非常简单，却非常高效！建议在这些招式中选择任意一个或几个，然后立即将其运用于自己的工作和生活中。如果事实证明这些招式是有效的，则向自己的团队成员分享和推广这些招式，从而实现自己和团队成员的共同成长。

8.4.1 确定在做的事情符合自己的目标

在谈论高效时间管理之前，我们必须十分清楚自己的目标是什么，并且在行动及时间安排上始终瞄准自己的既定目标。从另一个角度来说，最糟糕的时间管理，就是花费很多时间做毫无意义的事情且做得十分圆满。

那么，应该如何高效地确定自己希望得到什么，然后相应地进行时间规划呢？可以通过 5 步法则来实现。

第 1 步：确定自己究竟想要什么

在现实生活中，我们有一个非常有意思的发现：一些人之所以比另一些人工作效率高、时间规划好，主要原因仅仅在于他们十分清楚自己的目标是什么。对自己想要什么及如何实现越清楚，克服拖拉和时间浪费就越容易，自己的工作也就完成得越顺利。

那么，怎么确定自己究竟想要什么呢？一种非常有趣且有效的办法是：利用面向未来的场景进行一次思考。

场景是这样的：

假设下周就是你的 100 岁生日，你的孩子和孙子们给你精心准备了一场生日聚会，他们会在聚会上致辞，对你过去的经历和成就进行总结。请思考，在这样一份致辞中，你希望听到什么样的总结？

基于笔者的个人经验，这是一个非常有趣且可有力帮助思考人生目标的“脑力练习”。当然，也可以把场景换成 1 年之后、5 年之后，等等。

同时，我们可以和老板坐在一起，针对自己的目标展开讨论。也许你会觉得，这难道不是每个员工都会做的吗？然而实际情况是，很多团队的成员都在日复一日地重复着那些毫无价值的工作，他们从未试图针对这些问题与自己的老板进行认真细致的探讨。

第 2 步：把自己的目标写下来

心理学研究证明，如果只把一个目标停留在脑海中，没有以书面形式记录下来，那么它就始终是一个空想，毫无生命力可言。相反，如果把自己脑海中的目标用白纸黑字写下来，这些目标就会变得更清晰、更具体，也会帮助自己创造一个看得见、摸得着的行动蓝图。

第 3 步：为自己的目标设定一个最后期限，并让大家都知道

任意一个目标或者决定，即使写下来，如果不设定一个最后期限，也不会让人产生紧迫感。从某种意义上来说，行动也就没有真正的起点和终点，所以自然而然地会拖延时间，工作效率也在不知不觉中降低。

在为自己的目标设定一个最后期限，并且让大家都知道（这非常重要）之后，出于心理学上“承诺和一致性”的压力，你就会在不知不觉中避免拖延，事情会完成得更快，工作效率也会提高。

下面讲讲心理学中关于承诺和一致性的小知识。

两位加拿大心理学家完成的一项研究揭示了人们在赛马场上的奇妙心理：只要一下注，下注者对自己所挑选的马获胜的信心就会立时大增。当然，这些马的实际获胜概率没有任何变化，马还是原先那匹马，站在跟原先相同的赛道上。只不过，在下注者的脑海里，一买下彩票，这匹马获胜的可能性就顿时变大了。这是承诺和一致在捣鬼。**承诺和一致性原理指的是，人一旦做出承诺，所做的行动都会跟承诺的事保持一致。**

第 4 步：将实现目标需要做的最重要的事情列出来

基于我们自己的实践，在列出实现目标所需要做的事情的时候，如果列出的任务有数

十件之多，那么最后往往无疾而终。因为毕竟有那么多事情要做或者没做，所以也就没有紧迫感了；另一种情况是，看到还有那么多事情没做，完成任务的信心也会随着时间的推移下降，最后也就不了了之。

因此，根据“魔数 3”原理，我们推荐的做法是，根据优先级从高到低只列出实现目标所需要做的最重要的三件事。这么做的好处是：因为只有三件事，所以人们会觉得更有信心去完成；同时，能够立即着手去实施，还能较快地看到成效，从而更有信心完成任务，进入一个正循环。

在最重要的三件事都完成之后，如果暂时没有达到目标，就再重复分解剩下需要做的最重要的三件事，如此反复，直到目标达成。

第 5 步：每天都做一些接近自己目标的事情

因为在第 4 步分解了最重要的三件事，所以我们可以非常容易地将这三件事列入每天的日程安排中。例如，阅读几页专业书籍、拜访潜在客户、对一个模块进行重构，或者和员工进行一次建设性反馈的一对一谈话。

当我们开始脚踏实地、一件一件地做关系到目标完成的事情，并且能够看到成效的时候，我们就会感到满足、愉悦和信心满满，从而投入更多的精力和时间在其中。在执行一段时间之后，就会发现事情一件接一件地完成了！

8.4.2 随时应用 80/20 法则

歌德曾经说过：“只要合理利用时间，我们就永远有充足的时间。”随时应用规划和分配时间的 80/20 法则，就是非常重要的时间管理方法之一。

80/20 法则可以说是时间管理和人生规划中最重要的工具之一，非常多的经济活动也受这一法则的支配，例如，20%的努力产生了 80%的结果，20%的客户带来了 80%的销售额，20%的工作体现了 80%的价值，等等。

把 80/20 法则运用到时间管理上，就意味着：如果你有 10 项工作要做，则做其中两项的价值比做其他 8 项加起来都要大！也就是说，虽然做每一项工作花费的时间可能都差不多，但是做其中一两项工作的价值是做其他工作的若干倍之多！

这也是我们建议仅仅为目标分解出最重要的 3 个任务的原因，因为这样就可以避免我们每天都把时间浪费在那些“无足轻重的多数事情”上。

要把 80/20 法则应用在时间管理中，我们就需要随时问自己这些问题：

◎ 这个工作是属于 20%高价值的部分呢？还是属于 80%低价值的部分？
◎ 对自己需要做的工作，我是否抵御住了先易（低价值的工作）后难（高价值的工作）的诱惑？
◎ 自己是否应该做一些减法？把目前工作中属于 80%低价值的部分减少一些，从而增加做另外 20%高价值工作的时间？

切记，无论做哪种选择，久而久之，这些都会成为我们的习惯。因此，如果我们习惯性地选择先做容易的低价值的工作，后做较难的高价值的工作，就会很快养成这样的习惯，尽管这并不是我们所希望的。所以，在日常工作中一定要时时保持警醒，多问问自己上面这些问题，确定自己把时间更多地花在了 20%高价值的工作上。

经验表明，只要想到自己即将开始一项重要的工作，我们就会动力十足，不再有片刻的迟疑。事实证明，完成一项重要工作的时间往往和完成一项琐碎工作的时间类似，但是完成高价值的工作，可以给我们带来较高的自豪感和满足感；而完成一项琐碎的工作，只能让我们有极低的满足感。因此，为了进入正向循环，请随时抵御“先易后难”的诱惑。

时间管理实际上是对人生的规划和管理。做事有方的人往往懂得先处理最重要的事情。所以，你可以考虑在办公桌的案头贴上一张纸条：“下定决心，从今天开始把更多的时间放在 20%高价值的工作上，因为，这能让我不同凡响！”

8.4.3　创造大块时间

在确定了自己的目标，并利用 80/20 法则确定了属于 20%高价值的工作之后，接下来是不是可以达到高效时间管理的目标了呢？答案是：还需要有效执行。

如何利用时间管理工具助力有效执行呢？有人说：“如果把精力集中在有限的几个目标上，前途就不可估量”。而“创造大块时间”这个高效时间管理工具，可以让我们把精力集中在利用 80/20 法则梳理出来的有限几个高价值的工作上。因为许多重要的工作，例如软件架构设计、代码开发、需求梳理等，都需要不间断地完成。

事实上，许多成功的销售人员每天都会抽出一段固定的时间给潜在客户打电话。虽然这属于 80/20 法则中高价值的工作，但是谁也不会觉得这些电话拜访每次都是愉悦的，但是成功的销售人员不会推迟这项工作，即便自己不喜欢，他们也下定决心：每天都雷打不动地给客户打电话，这就是在创造大块时间。

同样，许多成功的技术团队管理者每天都会抽出一段固定的时间，来阅读行业报告和商业分析报告，以便更好地了解市场的反馈。许多人也会每天抽出 30 ~ 60 分钟用于锻炼，或者在每晚睡前阅读精品图书，这样日积月累，既保持了身体的健康，也阅读了大量的优秀书籍。

要成功实施“创造大块时间”，从而有整块时间处理重要得多的任务，关键在于提前对每一天的工作进行有效安排。下面介绍一种被实践证明非常有效的对一天的工作进行划分从而获得大块时间的方法：每日时间分区法。

图 8.2 是每日时间分区法的一个示例，可以看到，一天的时间被划分为早晨、日间及晚上三个主要分区；同时，将每个分区都根据不同的时间段划分为不同的子分区。这样的划分来源于最新的脑科学研究：虽然早晨的 1 个小时和午间的 1 个小时有同等的时长，但是由于每个人的能量水平和大脑活跃水平不同，不同时间段的效率、创造力和能量都不尽相同。因此，为了更好地利用每天的时间，以及通过合理安排划分出大块的时间用于工作和创意性事务，我们需要基于时间和能量这两个维度对一天的时间进行规划。



图 8.2

基于时间和能量这两个维度，我们可以把一天划分为若干大分区和子分区，并且为每个分区都安排最适合的工作。例如，在如图 8.2 所示的分区中，日间被划分为专注工作区（Focus Zone）、能量补充区（Energy Zone）及创意工作区（Creativity Zone）三大分区；每个大分区又被划分为若干子分区，并安排出了整块的工作时间。

研究证明，当人在工作中被打断以后，平均需要 15～20 分钟才能恢复到被打断前的效率。因此，为了更好地实施每日时间分区，一个最佳实践是：主动通知其他人不要打扰，例如在即时通信工具中设置状态为“正忙”；或者切断干扰源，例如把手机设置为静音状态，从而不间断地充分利用大块时间进行高效工作。

对一个技术团队的管理者来说，高效时间管理是事业成功的关键。所幸，我们的经验证明，上述三个高效时间管理招式都是可以通过反复实践来学习和掌握的。一旦定下明确的目标，再通过 80/20 法则确定关键任务，并为每天的时间分区都划分出大块时间的习惯，就朝着成功迈出了坚实的一步。下定决心，每天都实施这些工具和规则，直到它们成为我们的“第二天性”！同时，请帮助团队成员养成类似的高效时间管理习惯，这样的团队一定能够成为高绩效的团队！

8.5　遇到“不服管”的员工怎么办

保罗（Paul English）是专注于旅游领域的搜索网站 Kayak.com 的创始人，因为独创的“五字员工反馈法（Five-word Feedback）”而闻名。在快公司网站（fastcomapny.com）对他的采访中，保罗讲述了发明“五字员工反馈法”背后的有趣故事。

在创办 Kayak.com 之前，保罗是马萨诸塞州 InterLeaf 软件公司的职业经理人。他手下有一位绩效不佳的员工，保罗每个月都会给该员工做一次绩效评估谈话，以期让该员工的绩效恢复到正常水平。然而，保罗失望地发现，这些频繁的绩效评估谈话只是被该员工当作一次又一次无趣、官僚且引发双方对立情绪的程式化活动，对其绩效提升没有一点作用。

因此，保罗决定换一种方式，与其给员工一些缺乏具体事实依据、空泛的绩效评估，不如给员工提供一些非常具体的他做得很棒的事情，以及他搞砸了的事情的反馈；同时，为了避免让事情变得过于复杂，保罗决定只用五个单词来总结这些反馈。例如，在一次反馈中，保罗为这名员工写下的五个单词是：（1）快速响应（fast）；（2）专注聆听（attentive）；（3）缺乏对其他同事的信任感（untrusting）；（4）太过谨小慎微（too-cautious）；（5）专注

技术（technical）。同时，保罗还会面对面向员工解释这些反馈单词并展开讨论。

在保罗用这种非正式的、基于事实的反馈方式和该员工进行沟通之后，有趣的事情发生了：该员工的绩效开始提升。15 年之后，那位员工告诉保罗，他仍然保留着当年的五字反馈——“这是我从我所共事的经理那里得到的最好的反馈！”

遇到“不服管”的员工怎么办？只要在百度上输入这个问题，就能搜到很多方法，这些方法大概会从如下思路中提供解决方案。

◎ 典型的技术思路：找出“不服管”的原因所在，针对原因找出解决方案，实施方案并解决问题。
◎ HR 思路：团队管理者先做自我检查，然后和不服管的员工进行个别谈话，如果再不服管，就要明确规章制度、各岗位职责及行为规范，明惩严戒，如果还没有效果，就需要“杀鸡儆猴”。
◎ 资深管理者的思路：第一，安排能“平衡”该员工的人，正所谓“一物降一物”；第二，设定明确的考核指标并进行考核；第三，论功行赏；第四，给予信任，并委以重任。

可以发现，这些方法的核心和出发点，都在于如何把“不服管”这件事搞定，而最后的目标，也是让员工从“不服管”到“服管”，从而完成组织的目标。这种思路当然无可厚非，然而，如果回忆前面对管理和领导力的对比，我们可以很容易看出这些方法都出自典型的“管理思路”，专注点在于搞定“不服管”这件事，而领导力的核心点在于“人”。

接下来看看如何基于前面讲到的技术领导力 4D 框架，来应对“不服管”的团队成员。

8.5.1 每个人都是不错的

如果遇到不服管的员工，而团队管理者内心没有坚持“每个人都是不错的”信念的话，就非常容易给这样的员工打上“不听话”“老油条”“恃才傲物”等标签，而员工一旦被打上了这样的标签，管理者就会不自觉地认为这些员工是“有问题的”。

同时，由于每个人都有敏锐的觉察力，所以只要团队管理者心中有了这些标签和认定，相应的团队成员就一定能够感受到。可见，如果在团队管理者和团队成员之间有了这样的“气场”，那么后续希望解决“不服管”这个问题的很多措施，效果都会大打折扣。

原因很简单，正如著名的诺贝尔经济学奖得主丹尼尔·卡尼曼在其大作《思考，快与

慢》中指出的：人类的思考有两种方式，较快的无意识的“系统 1”，以及较慢的有意识的“系统 2”。

◎ 常用的、无意识的“系统 1”依赖情感、记忆和经验迅速做出判断，它见闻广博，使我们能够迅速对眼前的情况做出反应。但“系统 1”也非常容易让人上当，因为它总是坚守“眼见为实”的原则，任由厌恶和乐观偏见之类的错觉引导我们做出错误的选择。

◎ 有意识的“系统 2”则通过调动注意力分析和解决问题，并做出决定，它比较慢，不容易出错。

正如丹尼尔·卡尼曼一针见血指出的：人总是有一种倾向，就是我们大脑中的“系统 2”很懒惰，往往会走捷径，直接采纳“系统 1”的直觉来判断结果。这就是著名的“最省力法则”：如果达成同一个目标的方法有多种，人们往往会选择最简单的那一种。

而在现实生活中，“最省力法则”最直接的一个体现，就是锚定效应。锚定效应是心理学领域非常重要的一个现象，指的是当人们在对某人或者某些事情做出判断时，非常容易受到第一形象或者第一次获得的信息的支配——就像轮船的锚一样，把人们之后的判断和决策固定在某个地方。

如果团队管理者在心中给员工打上了“有问题”“老油条”等标签和认定，则后续的很多判断和行为都会被这些负面标签和认定所“锚定”。“保罗失望地发现，这些频繁的绩效评估谈话只是被该员工当作一次又一次无趣、官僚且引发双方对立情绪的程式化活动，对其绩效提升没有一点作用”就是锚定效应在这个案例中扮演的“角色”的生动写照。

相反，如果团队管理者心中秉持“每个团队成员都是不错的”的信念，同时坚信“每个人当下所做的决定都是最好的选择”，以及“人们拥有达到其目标所需的所有资源”，那么团队管理者被“锚定”的思维就是一个正向和积极的思维，而团队成员也会感受到这些，从而为后续采取的措施发挥效用打下坚实的基础。

8.5.2　建立亲和与信任感

《团队协作五大障碍》这本管理学名著通过主人公凯瑟琳的故事，生动展示了一名优秀的领导者是如何化解团队协作障碍的。

57 岁的凯瑟琳接替杰夫担任决策科技公司的 CEO 时，该公司上下一片哗然，因为凯瑟琳不仅过了创造力最强的年龄，也没有拿得出手的教育背景。凯瑟琳面对的是所有企业都会出现的问题：管理层互相倾轧，员工和上司离心离德，与竞争对手相比，他们掌握着更好更新的技术、更强大的董事会及更多的资金，尽管有这些优势，但是整个公司的盈利额和客户数量远远落后于对手。凯瑟琳，一个没有技术背景，却拥有促进团队协作超常天赋的蓝领主管，在众人怀疑的眼光中开始了她整合团队的历程。

在第一次举办全体管理层都要参加的会议时，她把会议举办地点选在了公司外的纳帕谷，换一个环境，也许能让所有人都有些启发。凯瑟琳在主持这次会议时，并没有如参会者所预料的那样从公司目前的困境或者描绘一个光明的未来开始，而是为这次管理层会议定下第一个话题：探讨个人经历方面的事。

凯瑟琳告诉主管们，每个人都要回答 5 个关于个人背景的问题，但问题本身不会让他们为难。她风趣地说：“记住，我想听你们儿时的经历，而不是你们现在的童心未泯。”

参会者一个接一个地回答了问题，包括家乡在哪里、家里有几个孩子、儿时的爱好、成长道路上的最大挑战和第一份工作等。

几乎每个人的回答都是其他人所不了解的，许多人也从中找到了共鸣。

坚冰开始融化，信任开始萌芽。

在这个案例中，凯瑟琳通过建立亲和与信任感来应对管理层互相倾轧、员工和上司离心离德的问题。这对我们应对“不服管”的员工有非常大的启示意义：需要在解决具体的“高难度”的“事”之前，从最重要的“人”出发，建立亲和与信任感！

为什么“建立亲和与信任感”是应对“不服管”员工的第 1 步呢？因为在团队协作的障碍中，导致其他障碍或者团队协作问题出现的根本原因，就是缺乏信任感。因此，在团队管理者和团队成员之间必须先建立起团队成员之间、团队领导者和成员之间的亲和与信任感，进而解决团队的其他问题。

那么，如何建立亲和与信任感呢？我们看看《亮剑》中的主人公李云龙是怎么做的。

在《亮剑》中，时任 386 旅独立团团长的李云龙，被上级安排赵刚作为政委。李云龙向来无法和政委很好地合作。而这一次，赵刚政委却让李云龙另眼相看：在刚刚结束的和日本鬼子的战斗当中，赵刚政委百步穿杨，3 枪解决了 3 个日本鬼子。等战斗结束后，回到驻地，便有了李云龙和赵刚的这一段对话：

李云龙：政委，老赵！

赵刚：团长，有什么情况？

李云龙：没什么事，咱们出去走走？

赵刚：走走？你没什么事吧，团长？

李云龙：别团长团长的，叫老李多亲切啊！你那次不说了吗，等仗打完了，我老李和你增进一下感情。

赵刚：我说老李啊，你不会又捅什么娄子了吧？

李云龙：看你说的，你把咱老李当成啥人了？我是真心地觉得，你小子挺有能耐的。大学生，这可真是文曲星下凡到咱们独立团了。

赵刚：这我得判断一下，这是夸我呢？还是骂我呢？

李云龙：你看，老赵，你这就没意思了吧？你们这些知识分子，就是心眼多。我可是真心话。老实说，你刚来的时候，我还看不上你。找机会我得把这小秀才给挤走。

赵刚：我看出来了，真够阴险的。

李云龙：你小子也不含糊，刚来的时候，不显山，不露水的。关键时刻，给我露了一手。一下子就把老哥我给装进去了。150 米的距离啊，3 枪，撂倒了 3 个鬼子。

赵刚：我在抗大学习的时候，有个教员，是射击高手。经常对我进行单独的射击训练，他认为我这个人啊，具备狙击手的潜质，通俗的说法就是神枪手。

李云龙：老赵，你可真有学问。以前我多有得罪，你可不许记我的仇。

赵刚：老李，这话老这么说就没意思了。我这个人啊，也有很多缺点，也需要改造。

李云龙：老赵，我得给你提个意见：你得学会喝酒。咱俩要当搭档，你想，咱俩怎么结合呢？就是一起喝酒，一起打鬼子嘛！

仔细分析李云龙这一段和赵刚政委“增进感情”的“教学”，我们可以发现李云龙采用了两种方法，这两种方法对于希望和“不服管”的员工建立亲和与信任感的团队管理者来说，同样有借鉴意义。

1. 工作之外的背景介绍

李云龙和政委“增进感情”，并不是通过讨论严肃和正式的话题完成的，而是通过和赵刚政委坦诚自己以前对政委“先入为主”的错误预设，以及神枪手这个话题，聊到了之前赵刚在抗大学习的经历。也正是这些工作之外的背景介绍，拉近了两人的感情，建立了亲和与信任感。而李云龙这个看似不经意的聊天，正是使用了建立亲和与信任感的管理工具：工作之外的背景介绍。

心理学研究表明，在影响信任感建立的因素中，相互之间的了解是很重要的。信任感建立在一定的相互了解基础之上。团队领导者和团队成员之间信任感的建立，也必须在一定的了解之后才会开始。而促进相互了解最重要的手段，就是对双方工作之外的个人背景进行了解。

这项方案的实践可以是在一次轻松的午餐时，或者是在第一次和新团队成员一对一谈话时或者见面会上，团队领导者和团队成员就工作之外的个人背景进行一些介绍和互动，例如自己的家乡、毕业的学校和专业、第一份工作、之前工作过的公司、爱好、家庭情况、个人的有趣经历等。在彼此介绍工作之外的背景时，团队领导者往往能够发现自己和团队成员间的某些相似性，比如毕业于同一所学校，或者都有一个可爱的儿子等。正如西奥迪尼博士在其《影响力》一书中揭示的：我们喜欢与自己相似的人，不管相似之处是观点、个性、背景还是生活方式，我们总有这样的倾向。而喜好正是建立信任感非常重要的助推剂。

针对“不服管”的员工，团队管理者也可以采用这一工具，作为融化双方“坚冰”关系的第一招。

2. 勇于展示个人的不足

我们注意到，在李云龙和赵刚政委坦诚自己之前对政委所抱有的错误预设，以及说“之前多有得罪，你可不许记我的仇”之后，赵刚政委就回答道：“我这个人啊，也有很多缺点，也需要改造。”两个人都鼓足了勇气展示自己的不足，其关系也在不知不觉中拉近了很多。实际上，李云龙使用的就是建立亲和与信任感的另一个管理工具：团队领导者勇于展示个人的不足。

“展示个人的不足”听起来似乎充满了风险，并且和建立信任感的关系不大，事实上恰恰相反，这非常有利于在团队领导者和团队成员之间建立信任感。在实践中，笔者会在第一次和新团队成员进行一对一谈话时讨论这个话题：“你能介绍你的三个优点和三个还

可以做得更好的地方吗？我之后也会和你分享我自己的答案。”这么做，既减少了团队成员的压力，又为对方选择展示什么样的优点和不足留有余地，让团队领导者有机会逐步创建一种“在我们之间暴露不足不会有负面影响”的团队共识。这也是团队领导者与团队成员在日后的沟通中能够无惧冲突、大胆创新和承担风险的前提。

瑞典声田（Spotify）公司的 CEO 在打造“无惧犯错”文化时提到：既然错误不可避免，那我们只需比其他人更快地犯错并更快地改错，我们无须恐惧犯错。”这一切，都是建立团队信任感的基础，也是日后打造高绩效团队的基础。

在了解了对方的更多背景之后，可以继续深入：

我们已经通过相互分享工作之外的经历，了解了对方的一些背景信息，我想，我们第一次发现了彼此很多之前都没有想到的内容，比如我们发现学的是一样的专业。那么接下来是否可以进行一个更有趣的“冒险”，尝试向对方分享我们愿意分享的两个强项和两个弱点。因为在相互了解了对方的强项和弱点后，在今后的合作中才能更好地发挥各自的强项，同时避免一些无谓的猜测。不知道你是否愿意？

作为团队领导者，可以坦诚地分享自己的一些强项和弱点，并委婉地邀请团队成员也进行分享，分享的内容以双方愿意暴露的程度决定。实践证明，即使双方分享的是无关痛痒的内容，但是敢于坦诚地与团队成员分享自己的强项和弱点，就是建立团队领导者和团队成员之间“基于暴露弱点的信任感”的开始。

8.5.3　从绩效评估到建设性反馈

在初步建立了和团队成员间的信任感之后，团队管理者是不是忍不住想处理“不服管”这件事了呢？回答：是，也不是。

回答“是”，是因为我们建立亲和与信任感的目标是让“不服管”的员工能够更好地实现个人成长和公司目标的双赢；回答“不是”，则是因为我们还得从“人”的角度寻找最佳解决方法。

因此，我们建议团队管理者此时采用下面两种方法。

1. 用“增长式思维”设定成长目标

对于技术专家出身的团队管理者而言，遇到“不服管”的员工，首先想到的都是寻找“不服管”的根因，然后针对根因找出解决方案，这和针对技术问题如软件缺陷所使用的

“5 个为什么（5 Why）”方法的思路一致。我们不能说“把解决技术性问题的方法和思路用于解决人的问题”是不对的；但是基于大量的经验，用解决问题的思路解决人的问题，收效并不乐观。

因此，解决人的问题，必须采用不同的方法。我们通过大量的实践找到了一种好方法——增长式思维（Growth Mindset）。

增长式思维是由斯坦福大学的卡罗尔·德韦克教授提出的，卡罗尔·德韦克教授将人的思维分为两种：一种是增长式思维，另一种是固定式思维（Fixed Minset）。在她看来，一个人在拥有增长式思维时，将乐于接受挑战，并积极扩展自己的能力，这也是未来发展最需要具备的能力。

那么，如何把“增长式思维”用到解决“不服管”的员工这个案例中呢？在比利时知名企业顾问、演讲家和企业培训师路易斯·卡夫曼所著的《不懂带人，你就自己干到死》一书中，就展示了一个利用“增长式思维”解决人的问题的有趣例子。

金斯利先生拥有一家非常成功的塑料公司，但是他面临一个难题，在董事会中占有重要席位的女婿和公司的 CEO 长期以来关系紧张。最近，发生了一件让他们二人爆发冲突的事件：女婿在没有咨询 CEO 的情况下购买了一套新的电脑办公系统，为此，CEO 威胁要辞职，并说出了“有他没我，有我没他”这样的狠话。金斯利先生因此对女婿非常生气，但又无可奈何，因为女婿从来都是独断专行的。怎么很好地处理这个问题，成为金斯利先生面对的一个棘手的问题……

这一次，金斯利先生决定采用“增长式思维”，与女婿和 CEO 一起进行一场开放性和建设性的谈话。

这次谈话是从一个“有力的问题”开始的。当 CEO 和女婿都坐在他面前之后，金斯利先生问了他们一个问题：“我想说清楚，对我女婿的行为，我既不欣赏也不赞同。但是，那件事已经过去了，我们也不用再在这件事上纠结。今天，我想和你们谈一些别的事情。我想请你们想一想这个问题：‘你们认为，公司里比较好的方面是什么？你俩在过去半年是如何合作来促成这些好方面的呢？’”

听到这个问题，CEO 和女婿都非常吃惊，因为他们都以为金斯利先生会大发脾气，也准备好了各种反击的说辞。但是当金斯利先生提出这个问题的时候，他们都同时领悟到了在这个问题中所蕴含的智慧。不难想象，接下来的谈话与以往不同，是一场非常富有建设性的谈话，CEO 和女婿也达成了之前难以达成的共识。

这个例子充分体现了利用“增长式思维”解决团队中人的问题的核心思路。

核心思路一：面向解决方案。实际上，“面向解决方案”的思维尤其适用于技术团队。因为对于工程师而言，长期的工程训练形成的思维定式，都是首先专注问题，然后发掘问题产生的原因，最终获得解决方案。但是，正如我们在上面金斯利先生的例子中看到的一样，如果仅仅关注问题，以及问题发生的原因，那么在处理团队中人的问题上，得到的往往还是无休止的争吵，而不会推动事情的解决。

与之相反，在“面向解决方案”的思维中，更多地聚焦于“尽管有问题，但什么是有效的”“尽管有问题，但希望在接下来的一个月或一个季度达到什么样的目标”。

这里，核心思路的转变就是：在人的问题上，不纠结过去发生了什么、是什么原因，而更多地面向未来，定义我们希望达到什么目标，以及有什么可能的解决方案。

核心思路二：充分利用“有力的问题”引发人的改变。“有力的问题”这一工具，虽然看似只是简单地让团队领导者和团队成员的沟通方式从陈述句（下达指令）变为疑问句（通过发问激发团队思考），但是获得的效果是非常显著的。针对一个期望解决的问题（例如制定团队愿景），团队领导者可以通过一系列有技巧的发问，即“有力的问题”，充分挖掘团队成员浅层次的意识和深层次的潜意识，从而达到激荡思维、激发创意、获得高质量解决方案的目的，而其核心在于如何问出“有力的问题”。

一般来说，“有力的问题”往往有如下特征。

（1）简短，直接，切中要害。与其他交际性的谈话不同，“有力的问题”无须拐弯抹角，而是直奔主题，试图用最少的词句直击问题的核心。

（2）能鼓励被提问者打开话匣。“有力的问题”是开放式的问题（例如 5W1H 问题：What、When、Where、Who、Why 和 How），因而能够让被提问者有空间更好地展开回答，从而有更多的机会获取更多的信息，以及激发更多的创意和思路。与此相反，封闭式问题（例如“是或不是”的问题）无法鼓励被提问者更多地去说，从而丧失了激发创意的绝好机会。

（3）能促使被提问者深入思考。一个出其不意的好问题往往会让被提问者一时语塞、沉默思考。研究表明，与滔滔不绝地说相比，沉默思考更有助于激发创意和思路。例如，“能告诉我更多吗？”“还有吗？”“还有什么缺失了吗？”都是能够促使被提问者进一步思考甚至一时语塞的“有力的问题”。“有力的问题”之所以有力，是因为它激发了被提问者深层次的潜意识。

一个脱口而出的答案往往来自浅层次的意识，或者说直觉，这也是我们最习惯的思维模式。重复这样惯常的思维模式，是无法激发创意和思路的。而激发深层次的潜意识，目的就是将我们存在于潜意识中的更丰富的信息连接在一起，从而产生新鲜的想法。

例如，针对那些所谓“不服管”的团队成员，下面这些问题就是用来激发深层次的潜意识的“有力的问题”。

◎ 想象一下，如果 1 个月之后，你对工作的投入度有了明显的提升，能够更好地和团队协作完成项目的目标，那么是因为谁做了什么样的改变呢？

◎ 如果我们的目标是让你在团队中有更大的成就，而你、我及团队的其他人都做了改变，你觉得会是什么改变？我们如何让这些改变发生，从而让你在团队中有更多的成就和投入？

◎ 回想一下你投入度最高的团队，如果能够做三件事，让我们的团队也成为你投入度最高的团队，你觉得会是什么事？

事实证明，当我们使用这些问题去提问时，绝大多数被提问者都会陷入沉思。但沉思之后的答案，往往充满意外的惊喜！所以，面向未来的问题比纠结过去的问题更有力。

2. 用建设性反馈助力快速成长

很多团队管理者在做到前面这件事情之后，就开始静静地期待奇迹的发生了。然而，他们忽略了一个为这个奇迹的发生添加最有效的“助推剂”的机会，这就是“建设性反馈”发挥神奇作用的地方。

还记得保罗的例子吗？保罗所用的非正式的、基于事实的反馈方式，就是团队管理者可以用到的另一种有效管理方法：建设性反馈。

“建设性反馈”中的反馈来源于控制论中的回馈概念，指的是将系统的输出返回到输入端并以某种方式改变输入，进而影响系统功能的过程。对于一位团队管理者来说，提供反馈的过程就是依据对事实的观察，将团队成员行为的影响或者效果以客观的方式提供给团队成员，以期对员工的行为产生影响（保持有效的行为或者改变无效的行为），从而达到改进绩效的目标。

然而，我们常常发现团队管理者会把“绩效评估”和“建设性反馈”混为一谈。实际上，绩效评估指的是通过系统的方法和原理测量员工在职务上的工作行为和工作成果，从而对员工做出综合评价。绩效评估的结果会直接影响到薪酬调整、奖金发放和职务升降等诸多员工的切身利益。表 8.1 对比了绩效反馈和绩效评估之间的区别。

表 8.1

绩效反馈	绩效评估
提供信息	提供评价
更多使用动词和名词提供客观的信息	更多使用形容词和副词做出评价
较为客观	较为主观
较为频繁（例如每天、每周）	较为不频繁（例如每个季度、每年）
非正式	较为正式

为了让绩效反馈更加有效，越来越多的团队领导者开始使用“建设性反馈”这个有力的工具。每个人都喜欢被称赞，很少有人喜欢被批评，这也是绩效评估无法达到预期的改变员工行为目的的原因。例如，当一名团队领导者告知团队成员，“你最近工作非常糟糕”或者“我不喜欢你最近的工作汇报”时，员工并不能从中获得如何提升的具体信息，相反，批评还会使员工产生抵触情绪。而“建设性反馈”提供了一种更加有效的基于事实、关注解决方案的双向沟通机制，目标是让员工更加主动地改变行为以提升绩效。

在实践中提供“建设性反馈”可以遵循下面的五步“公式”进行：

建设性反馈 = 明确目标（“我希望和你讨论_____”）+
观察（“我观察到____”）+ 影响（“这么做的影响是____”）+
要求（“我期望今后你能____”）+ 讨论（“你觉得呢？____”）

第 1 步：明确此次“建设性反馈”的目标

开门见山、言简意赅的“建设性反馈”的主题，可以让接收反馈的对象无须猜测这次谈话的原因，也能让接下来的谈话更加聚焦。

第 2 步：具体描述观察到的事实

建设性反馈最重要的基石在于具体的事实，所以在描述事实时应该尽量清晰地指出时间、地点，以及发生了什么。更重要的是，一定要使用第一手的观察，尽量少使用从其他人那里获得的观察，从而避免反馈和事实出现偏差。

第 3 步：阐述行为所产生的影响

在描述了观察到的事实之后，接下来需要阐述这些行为所产生的影响。这里所说的影响既包括行为所带来的可观察到的结果，还包括行为所影响的人的反应。只有当获得反馈的一方充分了解了他们的行为所产生的影响之后，才会为讨论应该如何对行为进行改变打下基础。特别需要注意的是，这里的影响是一个中性词，既包括正面的影响，也包括负面的影响。

第 4 步：清晰描述期望的行为

“建设性反馈”的最终目的还是希望在事实和影响的基础上，得到反馈的一方做出行为的改变。这里的改变既包括对正向影响行为的继续保持或者发扬，也包括对负面影响行为的改进。因此，在描述了行为和影响之后，更重要的是从面向未来的角度清晰描述自己的期望，并提供切实可行的改进行为以达到目标的建议。同时，为了提高被反馈者的接受度，期望的表述应该尽量从面向未来和解决方案的角度表达，而非面向过去和问题的方式。

第 5 步：给对方机会阐述他的想法

“建设性反馈”是一个建立双向沟通及激发建设性讨论的好机会。因此，在陈述完事实、影响和期望之后，还需要给接受反馈的一方阐述自己想法的机会，从而让双方就期望的行为及如何达到期望开启一段建设性的讨论。

那么，针对利用“增长式思维”和“有力的问题”一起确定了改进提升目标和行动计划的员工，下面是一个给予建设性反馈的例子。

小王，我希望和你谈一谈之前确定的提升你在团队中的参与度的计划。我观察到，基于你自己的兴趣，你现在已经开始承担团队中持续集成系统的升级这项工作，而且目前已经在一周之内完成了 Jenkins 环境的搭建。这个积极的进展，为我们在 3 季度顺利完成 DevOps 的测试原型这个目标打下了坚实的基础；也向其他部门展示了我们团队在持续集成领域的业务能力，赢得了团队内部其他同事的一致好评！

小王，我期望你能够保持目前这种勇于尝试的精神，在不断提升自己能力的同时，确保按时和高质量地完成 DevOps 环境测试原型的搭建工作。这也是咱们之前谈到的，提升你在团队中的参与度，实现个人发展和团队目标达成实现双赢这个目标的重要举措。

小王，你自己的感受和反馈如何呢？

在使用上面这几个管理工具之后，你会发现，即使是那些之前看起来“不服管”的员工，也会有如下变化。

◎ 在团队管理者和团队成员之间会有更多的亲和与信任感，从而让“对立关系”变成“伙伴关系”。
◎ 团队成员会更多地把可能存在的问题看作不断改进的机会，而不是争论谁对谁错的起因。
◎ 团队成员会更加聚焦“如何解决问题”，并且受到“有力的问题”的启发，会有更多有创意的解决问题的思路和方法。
◎ 团队成员更易于接受你的反馈，无论是做得好的事情还是有待提升的领域。
◎ 曾经让你头疼的气氛紧张的绩效评估谈话变成了面向未来的更富有建设性的讨论。
◎ 最重要的，在关注的重点从问题（绩效评估中的评价部分）转变为面向未来的解决方案（绩效反馈中的期望部分）之后，员工将有更大的可能性做出行为的改变，从而达到提升绩效的目的。

8.6　如何处理冲突

在技术团队管理的过程中，冲突是我们永远无法回避的话题。是否具备处理冲突的能力，往往被当作评判一个管理者是否合格的标准之一。冲突也一直是重要的团队协作变量，所以团队间冲突与绩效的关系也是人们一直在探讨的话题，然而人们很少探讨冲突管理行为，这就导致大家忽视了在冲突中潜在的积极意义，所以让我们一起来正视冲突，选择策略并做好冲突管理吧！

8.6.1　了解产生冲突的原因

一般来说，产生冲突的原因有如下几种。

1）目标理解不统一

我们将这种冲突称为目标冲突，产生的原因往往是团队对目标理解不一致，制定了非协助性 KPI，大家在实施过程中为了完成各自的 KPI 而出现相互干扰的情况。面对此类冲突，只需通过深入沟通、仔细分析达到对目标理解的一致性就可以解决。

2）多种技术方案选型

在大型技术团队中，团队成员的经历不同、对问题的解读不同，在技术方案决策上往往出现此类冲突，但是这种技术上的冲突往往具有积极的意义，可以帮助大家开阔视野。作为管理者，我们只需做好正确的疏导和决策就能化腐朽为神奇。

3）多项目并行引起资源冲突

由于项目紧急或排期不当，一个团队往往会出现同时承接多个项目的情况，这会导致团队非常疲倦，且因为不能专注于某个具体的项目，导致结果与预期背道而驰。此类冲突出现的主要原因是项目计划不当和应用资源不当，因此解决方案顺理成章地就是对项目进行更合理的规划，需要跟项目的提出方不断进行更深入的沟通，对项目做出更合理的优先级排序，以便充分利用有限的资源，让项目更有序地进行。

4）项目进度冲突

在实际的项目实施过程中，需求调整、方案变更时有发生，但项目完成时间又是事先敲定的，或因为类似“双十一”的特殊促销活动不能随意修改时间，就会经常出现研发人员工作时间与项目进度之间的冲突。大量的加班和不断变更的方案会让研发人员身心俱疲，因此需求提出方一定要对项目的核心目标进行认真、仔细的思考，并把每一步的逻辑和细节进行充分拆解。同时，项目的承接团队在需求评审时也一定要结合技术方案和业务需求进行更仔细的思考，以免在实施过程中出现较大的需求和方案变更。

5）团队成员的情感因素

团队成员间因情感因素发生冲突主要有如下原因。

◎ 个性不同：大家的沟通方式、方法差异导致沟通不愉快而发生冲突。
◎ 团员间的偏见：表现为对他人的观点不赞同，甚至一言一行都看不惯，这样势必发生冲突。
◎ 价值观不一致：表现为因经历不同、背景不同、观念差异而产生冲突。

此类冲突处理起来最为棘手，要求团队管理者具备一定的心理学及人力资源管理经验，能够在出现此类冲突时正确化解，以及对人员组成进行及时调整。

8.6.2 正确看待冲突

在大多时候，大家都认为冲突只会产生不好的结果，但是世间事都有两面性，冲突同

样有积极的意义。例如，上述“多种技术方案选型冲突”就可以通过技术讨论拓宽视野，弥补冲突各方的认知盲区。我们将冲突产生的积极意义归纳如下。

◎ 暴露问题，及时熔断。
◎ 促进了解，加快团队磨合。
◎ 弥补不足，拓宽视野。
◎ 产生竞争，培养主动思考能力。
◎ 信息流动，引导创新。
◎ 团队涅槃，完善管理。

依据我们的经验，在各种技术会议上最恐怖的不是大家的激烈争执和讨论，而是死亡般的沉默，大家之所以不提问题往往不是因为技术方案完美无缺，而是因为完全不思考或根本没有听懂，所以技术管理者适当地发起讨论，引导并激发大家狼性竞争，也是管理行为艺术的表现。

消极面可能是大家想到或遇到最多的冲突表现，凡是影响团队存亡、技术发展、目标达成和团队团结的冲突，都需要加以干预和控制。这里还是以“多种技术方案选型冲突”为例，当双方对这种冲突僵持不下时，就会威胁到项目的进度，这就是冲突的消极表现。此时管理者需要及时介入并控制，以确保项目顺利完成。

8.6.3　处理冲突

管理者处理各种冲突的基本原则就是客观和公正，如果丢失了这两个原则，那么所有的策略基本无效，甚至会让事态变得更加剧烈和不可控。

1）强调目标和项目的意义，建立共赢意识

让冲突中的每个人都认识到大家是在一起完成一件有意义的事情，特别是在发生“情感冲突”和“项目进度冲突”时，明确冲突对目标的影响，唤醒大家的共赢意识，把冲突的各方从不理智的漩涡中拉出来。

2）搁置冲突，冷处理

管理者应该对冲突的大小和影响范围有清晰的判断，如果只是很小的冲突且不会对团队及目标有任何影响，则可以采用不予理睬的冷处理方式，让冲突自行淡化并消失。

3）管理协调，双方妥协

当发生“项目并行人力资源冲突”，特别是需求双方属于不同业务线时，管理者需要将双方组织在一起，对项目时序进行合理调整，如果双方项目都必须同时进行，就需要对共有资源进行重新分配，而这种分配可能会造成交付时间延期。在这种情况下双方可能需要在项目交付时间或项目时序上做出让步。

4）建立制度，法治管理。

在冲突处理中，人治管理的最大问题在于管理者自身的情感控制和对事态严重程度的判断，这些稍有闪失就会对团队及目标产生消极的影响，所以最好完善制度，给团队划底线——让所有人都清晰地认识到底线在哪里，以及触碰底线的后果。

5）职务介入，强制执行

当事态已经发展到风险预警阶段时，管理者就需要快速做出决策。比如在上面提到的多种技术方案选型冲突中，如果双方仍然各持己见无法让步，管理者就需要快速做出决策和命令，确保项目顺利进行下去。

8.7 引导员工主动工作

相信没有任何一家企业希望自己的员工对工作敷衍了事，像木偶一样需要不停拉动才能工作。特别是在当前竞争如此激烈的市场环境下，企业更需要能主动工作和主动思考的员工，那些员工如木偶般工作于企业来说是一种极大的风险。

很多时候，我们的管理者把员工被动工作的原因归结为个人因素，例如员工情绪、工作动机、态度甚至个人素质等，这种被动状态下的工作效果可想而知，企业也不会对这样的员工有任何奖励。

殊不知员工的行为其实是环境的产物，除了需要员工自己努力，营造适当的工作环境也尤为重要，所以雇佣双方都应该积极想办法把工作从被动变为主动。

定义正确的角色及职权范围，是保障企业自我优化和成长的关键。笔者和很多企业管理者都有过沟通，在沟通过程中，几乎每位管理者都有相同的状态——疲惫。这种疲惫源于团队缺乏主动工作的思想，很多工作的决策、优化的思考都会落到管理者身上，其他人永远都在自己的“盒子”里工作，所以大家不难发现“盒子”成了阻碍大家主动工作的“牢狱”。

那企业里的“盒子”又是什么呢？在企业实际运行的过程中设置了很多部门和职位，

比如总监、经理、主管、普通员工等，在日常工作里，每个上级的职位都被赋予了“家长”的角色，当这个“家长”迫切追求结果而越俎代庖时，就会让员工身处自己职位的“盒子”中，丧失了主动工作和思考的能力。

所以正确的方式是企业定义好员工的角色，让员工清晰地认识到自己应该承担的责任和能决策的范围，而不是把自己放在职位这个“盒子”里。例如，在一个技术团队里有主管、前端工程师、Java 工程师这几种角色，那么主管的角色应该被定义为企业的路由器，肩负起拆解企业目标、团队发展、组织战术会议及团队间沟通的责任，而不是关心前端工程师或 Java 工程师的代码怎么写才能更好地实现目标，或者怎么优化这些代码才能更满足用户的体验。前端工程师和 Java 工程师的角色就应该被定义成目标的落地者，所以这个角色的职责应该不停地思考怎样才能更好、更快地实现目标，考虑自己的工作对其他角色的影响，以及其他角色的工作对自己的影响等，而不仅仅是完成既定的代码工作。

我们可以通过以上方式把每位员工都放在一个被定义好且被赋予了明确职权范围的角色上，让其能自主工作而不是进行家长式的管理。布赖恩·罗伯逊在其《重新定义管理》一书中提到的“合弄制（Holacracy）”就是一种让员工能够自主工作的很好方式。合弄制有时也被称为全体共治，顾名思义，就是由角色来承担工作的管理系统。一项工作被看作一个“角色”，同一个人可以选择承担不同的角色，和其他人配合完成工作，按照角色分配权力。合弄制被认为是一种无领导管理方式，它将公司组织架构去中心化，将由人定义工作角色转变为围绕工作来定义，并且经常更新。

举个例子，我们的身体组织和运行方式，就是某种意义上的“合弄制”。人体的功能运转并非按照自上而下的指挥模式进行，而是一种分散系统：各部分独立运行，整体形成一个有机网络，每一个部分如细胞、器官及器官系统都能够自主接收、处理和输出信息。因此，当身体某个部位的白细胞发现了一种炎症时，它就无须把这个信息送到大脑，再等待指示，因为这样会延误抵御疾病的最好时机。实际上，白细胞就是“合弄制”中承担了抵御炎症这个“工作”的“角色”，它可以自主决定在发现炎症的时候产生抗体而无须等待大脑的指示。这样的运作模式，也保证了我们身体的正常运转，也正是当今企业的运转方式。

赋予员工做事的意义和目标，是打造高产出团队的基础，让员工有机会参与到宏伟的事业中，并且设置可实现的目标让员工报以希望地工作。虽然如此，在企业管理过程中，员工的个人需求和企业的需求往往没有办法完全达到一致。管理者要充分理解和尊重员工的个人需求，知道员工能承受的极限在哪里，而不是一味地采用压制的管理方式，把员工变成俯首帖耳的工作机器，很难想象这种环境下的员工怎能产生工作主动性和创造性。

因此，企业各阶层管理者需要在工作开始前跟员工进行充分沟通，让员工充分理解所做事情对于企业的意义。和员工一起制定合理的绩效目标，让双方在目标及需求上达到一致，才能营造一个让员工自主工作的环境。

在员工完成目标的时候，企业管理者应该对此做出及时的反馈，需要让员工更清晰地认识到自己对于企业的价值，我们将这种管理的方法称为“价值管理”。正如前面提到的，要营造适合主动工作的环境，一定需要跟员工一起制定合理的绩效目标。在制定目标时可以把目标从易到难地拆解成几个小的阶段，让员工能在阶段性的完成中收获到成功所带来的喜悦。

因此，在员工完成每个目标的时候应该都及时地给予反馈，例如对好的赞许、对不足的鼓励及对困难的帮助，给予员工更大程度的积极暗示。通过跟员工分享结果数据，让员工能直观地看到自己工作方式的改变给企业带来的价值，并在潜移默化中让员工成为一个能主动工作的人。

8.8 如何从带 10 个人到带 100 个人

拿破仑之所以成就了“千古一帝”的伟大传奇，是因为他非常聪明地发现了利用文化的强大力量去推动战略实现的秘密。

当年，在巴黎的一个房间里，拿破仑和所有将领坐在一起，讨论如何攻打俄国，拿破仑说：“我们现在讨论的进攻计划就是战略，但是，让 100 万大军心甘情愿地向莫斯科进军的是文化！”

那么，拿破仑是如何在军队中建立强大的文化来帮助他实现战略目标的呢？这就要提到拿破仑从一条绶带发现的秘密。“我有一个世上最奇妙的发现”，拿破仑曾说，“我发现人为了获得绶带愿意冒生命的危险，甚至宁死不惜。实际上，给我一条绶带，我会给你一支军队；给我足够的勋章，我能征服整个世界！”这里，拿破仑正是利用了军队的团队文化，将英勇善战和一个外化的团队形象标识“功勋绶带”连接在了一起。

对于技术团队的管理者而言，当其管理的团队人数从 10 个人转变为 100 个人的时候，在经历了初期的“权力更大”的欣喜之后，可能更多地面对困惑和迷茫。如果说管理 10 个人的团队，可以用管理 1.0 方法的话；那么，管理 100 个人的团队，就必须升级到管理 2.0。那么，管理 10 个人和管理 100 个人，有哪些主要区别？在管理上有哪些挑战？如何升级到管理 2.0？接下来一一详解。

8.8.1　在管理上面对的挑战

对于 10 个人的团队，团队管理者可以非常容易地叫出团队每个成员的名字，谁是什么性格，在能力和技能上的优势，在做什么事情，等等，都清清楚楚。在这个规模的团队中，技术团队管理者的核心要务就是完成关注点从“事”到“人”的转变，我们可以称这个阶段为技术管理 1.0。这一阶段的特点就是：拍拍肩膀（即建设性反馈）比严肃批评更管用，吃饭、聊天（即工作之外的谈话）比绩效面谈更管用。

在阿里巴巴创业初期，每当有人做得不错时，马云就给予其口头奖励“加寿”，比如“这件事做得不错，奖励加寿 200 岁”。有个能力超群的员工，最后居然成了“九千岁”，至今被阿里人传为佳话。这些，都是管理 1.0 的实例。

当团队规模扩大到 100 个人的时候，作为管理者，你会发现已经开始没有办法对每个人都叫得出名字了。这时，只采用管理 1.0 的方法和工具显然不足以应对。总结起来，从带 10 个人到带 100 个人所带来的管理挑战主要包括如下内容。

挑战 1：如何让团队统一目标和前进方向

所谓众口难调，当团队有 10 个人的时候，还比较容易协调大家的目标和前进方向；当团队人数扩大到 100 个人以后，每个人都会有自己的视角和利益，因而协调所有人保持一致的目标就变得尤为重要，也充满挑战。这也是拿破仑说出“让 100 万大军心甘情愿地向莫斯科进军的是文化”的原因。在巴黎那个房间里讨论进攻计划的将军们，就好比 10 个人的团队，他们很容易对战略和方向达成一致；然而，100 万大军，就好比这里所讲的 100 个人的团队，如何统一认知并执行战略，就是一个非常大的挑战。

挑战 2：如何让团队高效地交付产出

在通常的组织中，随着团队人数的增长，组织层级越来越复杂。在 10 个人的团队中，在大多数情况下，层级为 2：团队管理者和团队成员；在 100 个人的团队中，自然而然就会成立很多不同的部门，以及更多的层级，在这些不同的部门和层级产生之后，随之而来的是所谓的“部门墙”和“人浮于事”等问题。

管理大师埃德加·沙因（Edgar H.Schein）教授就描述过一个有趣的部门墙的故事。

汽巴-嘉基（Ciba-Geigy）公司是一个位于瑞士巴塞尔的大型化学制药跨国公司，公司邀请埃德加作为顾问帮助公司建立一种变革的氛围，从而让公司以更灵活的方式应对周边不断变化的商业环境。

和其他大型公司类似，汽巴–嘉基公司也由大量边界清晰的业务部门、区域部门和职能部门组成。令人欣喜的是，随着对这些部门、组织了解的深入，埃德加发现在组织中某些非常具有创新性的想法和事情正在发生。作为建立变革氛围的一部分，埃德加将他认为好的想法记录下来并交给了公司联络人，并恳请联络人将其分发给其他也会从这些好的想法中受益的业务部门经理。

然而，几个月之后，埃德加失望地发现，汽巴-嘉基的联络人压根就没有把那些好的想法分享给其他经理，而似乎不同部门的经理也没有太大兴趣相互分享和传递新的想法。造成这一切的原因，其实是因为在汽巴—嘉基公司的文化中，大家都有一个共同的假设：即每个经理所分管的领域，都是边界清晰的“私人地盘”，未经允许，大家都心照不宣地互不侵犯。因此，这就可以理解为什么联络人没有把那些好想法分享给其他经理，而且不同部门的经理人似乎也对跨部门的沟通兴趣不大。汽巴—嘉基的文化让这些行为像未经邀请擅自闯入别人家里一样粗鲁。

除此以外，在软件开发团队中，还有一个特有的问题，那就是功能团队（Feature team）和模块团队（Component team）之争。例如，对于一个开发电商 App 的 100 人大团队来说，如何划分内部小团队的职能？

一种方法是模块团队的思路：10 人小团队 A 负责前段，20 人小团队 B 负责后端，10 人小团队 C 负责测试，10 人小团队 D 负责持续集成。这样的好处是每个人团队看起来都职责分明，而且能够专注做一件事情，并且把这件事情做到最好。

另一种思路就是功能团队：将电商 App 划分为 A、B、C、D 四个端到端功能（例如搜索、商品组织、订单处理、内容发布管理等），然后每个团队负责一个端到端功能。这样优势在于每个团队都可以减少对其他团队的依赖，能够独立地交付可以供客户使用的软件。

然而，一直以来，这两种团队组织模式都在软件开发组织当中不断被争论，也有组织不断从一种模式转换到另一种模式。其背后的核心，都是希望解决一个问题：如何打破部门墙，加强部门协同合作，从而一起更高效地交付产出。

挑战 3：如何处理团队中的冲突和沟通问题

俗话说，“有人的地方就有江湖”，100 人团队的内部冲突一定比 10 人团队的要多。因此，如何处理 100 人团队内部形形色色的冲突，是 100 人团队管理者需要升级管理“程序”的地方。

同时，正如专门跟踪全球软件项目是否成功的 Standish Group 在 2013 年发布的数据显示，在 2012 年，仅仅有 39%的软件开发项目取得了成功。对于大多数失败的软件项目，问题的根源都在于团队缺乏透明和有效的沟通。当团队人数从 10 人增长到 100 人的时候，作为一个技术团队的管理者，你会更迫切地希望寻找到提升团队沟通效果的良方。因为此时，团队沟通的问题也成为你的高效管理心头之痛！

8.8.2　从管理 1.0 到管理 2.0

针对上面从 10 人团队到 100 人团队的三个管理挑战，这里有针对性地提供了从管理 1.0 到管理 2.0 的两种方法。

1. 通过“文化和价值观大讨论”充分发挥文化的指南针作用

“企业文化、愿景、使命以及价值观，看起来这些东西似乎都是专门为那些做不出或者卖不出好产品的企业和团队准备的，花费精力在这些 MBA 课堂上学到的东西只会影响到我们的正常工作，如果有时间的话，我们还不如好好执行我们的战略。”

这段话听起来很熟悉吧？这是《科学企业管理：成为成功创业公司的 24 个步骤》一书的作者比尔·奥莱特在其成立第一家名为 Cambridge Decision Dynamic 的公司时的想法。不出意外，比尔·奥莱特的第一家公司创业失败了。

在后来的总结中，比尔·奥莱特说道：“这家公司之所以失败，不是因为我们没有好的技术和产品，也不是因为我们没有客户，而是因为它不能成为一家可持续、可扩展的组织——我们没有任何有意义的目标，无法形成团队凝聚力，难以度过困难时期。总结一下，我们没有一种强大的企业文化。”管理大师彼得·德鲁克曾经用非常有意思的一段话总结了这一点：“对于文化来说，战略是早餐，技术是午餐，产品是晚餐。文化会吃掉后面的其他东西。”

企业文化是企业价值观的体现，可以引导员工做出正确的技术和商业决策，可以让员工知道如何与他人相处和互动，也可以让员工在关键时刻（moment of truth）做出最佳的判断。优秀的企业文化可以在员工之间营造出一种凝聚力。

因此，作为一个管理 100 人团队的管理者，充分利用文化和价值观的作用，就能帮助完成从“团伙”到“团队”的升级。

具体怎么做？为了打造切实可操作的企业和团队文化，先让我们从企业文化的定义入

手。企业文化，是一个组织由其价值观、信念、仪式、符号、处事方式等组成的特有的文化形象。因此，可以以这种特有的文化形象作为切入口，让团队成员通过充分的讨论形成一致的理解，从而让企业文化在团队中发挥如下作用。

◎ 胶水作用：让团队从一盘散沙变成有凝聚力和执行力的利益共同体。
◎ 地基作用：统一的文化是团队的“压舱石”，无论是在一路高歌猛进，还是在面临挑战和危机，都能让团队有稳定和坚实的成长基础。
◎ 标尺作用：共同的目标、一致认可的处事方式，都是在成长中团队成员衡量自己和他人工作的标尺。
◎ 罗盘作用：价值观指导了团队成员在工作中做出最优的决策，有了统一的价值观，无论环境如何变化，整个团队都能经由各个独立团队成员的决策整体朝正确的方向前进，就好似整个团队拥有一个共同的“罗盘”一样。有了这个共同的罗盘，“人心齐，泰山移”就能得以实现。

具体来说，经过验证，下面的方法和步骤能够帮助管理者在 100 人团队中打造统一的文化和价值观。

第 1 步：团队名称和形象标识讨论

为什么团队文化打造要从团队名称和形象标识开始呢？原因有两方面：首先，团队名称和形象标识讨论起来相对轻松，同时是团队文化中重要的组成部分，从这个领域开始讨论，能够让团队成员在轻松愉悦的氛围中进行想法的碰撞和磨合，为接下来的其他领域的讨论打下基础；然后，讨论团队名称和形象标识，能够帮助团队打造相互间的亲和感。正如西奥迪尼博士在《影响力》书中揭示的，人们都喜欢和自己有着共同特征的人，而讨论团队名称和团队形象标识的过程，就是团队寻找和确定共同特征的过程。

团队形象标识既可以包括团队 Logo、吉祥物、口号等大家共同珍视的东西，也可以包括团队故事，只要这些形象标识能够直接和明确地提醒和传递团队文化的核心部分即可。

第 2 步：团队价值观“内涵”和“外延”讨论

价值观是打造团队文化非常重要的一个环节。由于大多数企业都会有组织层面的价值观，因此在这一步实际上要做的就是如何让组织的价值观接上团队的地气，并且让团队对其“内涵”和“外延”都有统一的认知。

在实践讨论中，一种推荐的讨论方式是利用如图 8.3 所示的价值观“内涵”和“外延”四宫格组织团队讨论。

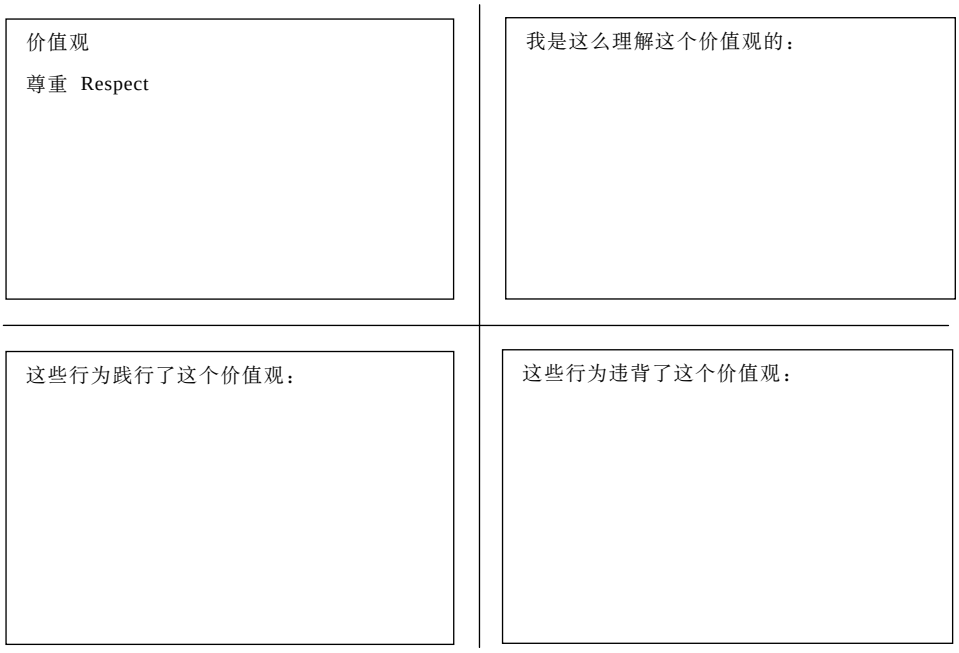


图 8.3

对于每一个企业，我们都可以准备一张大的白板纸，按照如图 8.3 所示划分为 4 个区域，同时给每一位参加讨论的团队成员发放若干张黄色便利贴。在讨论中，团队成员依次在便利贴上写上如下 3 个问题的答案。

◎ 问题 1：我如何理解这个价值观。
◎ 问题 2：哪些实际工作中的行为践行了这个价值观。
◎ 问题 3：哪些实际工作中的行为违背了这个价值观。

在每一位团队成员都把便利贴粘贴到相应的区域之后，主持人邀请每一位团队成员把自己所写的内容分享给团队的其他成员。接下来，团队将利用类似“点投票”这样的“发散—收敛”工具，为每个人分配固定的点数，让每个人将点数投给自己最为认可的讨论内容，从而实现对白板上的所有便利贴进行分类合并和收敛，最后利用共识式决策的方式，留下所有团队成员都认可的部分。通过这几步，就完成了团队对这个价值观“内涵”和“外延”的私人订制。

第 3 步：团队处事方式讨论

在企业文化的定义中我们看到，团队的处事方式同样是团队文化中非常重要的一部

分。然而，与团队名称、价值观等受重视程度不一样，团队处事方式往往会被团队领导者忽略。实际上，当我们对团队内部的低效决策或者不必要的内耗进行分析时，没有协同一致的团队处事方式常常就是造成问题的原因。

例如，对于软件开发团队，一个问题是：当我调用另一位同事开发的接口（API）进行编程时，我是否可以信赖他对接口的健壮性（Robustness）进行的设计和测试，这样我无须再进行额外的保护性编程。因为，冗余的保护性编程是一个典型的内部浪费。如果团队没有协调一致的处事方式，那将对团队的效率产生消极影响。

因此，我们可以在完成了对价值观的讨论之后，列举出工作中的一系列典型场景，让团队成员一起来回答："基于我们认同的价值观，我们在这个场景中应该如何行事？"例如，常见的场景包括：

◎ 如何面对失败；
◎ 如何处理工作中的冲突；
◎ 如何给其他人提供建设性反馈；
◎ 如何管理风险；
◎ 如何在创新和本职工作间达到平衡；
◎ 团队是如何庆祝"胜利"的；
◎ 团队共同的假设和期望是什么；

在使用了上述三步之后，我们的团队将有如下收获。

◎ 团队名称和形象标识；
◎ 对企业价值观的"私人订制"，即对企业价值观在团队中的"内涵"（是什么？）和"外延"（意味着什么？）的统一理解；
◎ 基于团队认可的价值观，团队协同一致的处事方式。

团队名称和形象标识能够大大增强团队成员与团队之间的连接，而针对团队特点"私人订制"的价值观，会成为团队中个人行为和决策的指南，而团队协同一致的处事方式，会成为价值观在实际工作中落地的最佳体现。有了这些助力，团队在打造高绩效团队上便又迈进出了坚实的一步。

2. 巧用 OKR

上面的管理 2.0 工具，主要关注通过团队愿景、文化和价值观这些思想层面的东西统一团队的认识。那么接下来，为了应对从管理 10 人团队到管理 100 人团队的管理挑战之

二——“如何打破部门墙，让团队整体有高绩效产出”，我们就需要有一个针对“事”的管理 2.0 工具了。

一说到“事”，大家往往会立刻想到“定目标，抓考核，严奖惩”，同时，浮现在脑海的一定有 KPI 这个工具。

但是就靠 KPI，能否实现“定目标，抓考核，严奖惩”的目标呢？在《谁说人是理性的》一书中描述了一个关于 KPI 的故事。

为了对授课教师进行评估，MIT 专门开发了一套非常复杂的计算“教学点数”办法。虽然这套复杂的“教学点数”的发明初衷是希望从多个不同方面去衡量和评估教师的绩效，但是很快就发现，这套评价系统本身却成为目标，而非手段。正如作者提到的，“虽然我从课堂教学中获得了很多乐趣，但是我却发现我越来越少地把时间花费在和学生们在一起，因为花费同样的时间，我可以找到其他事情获得更高的‘教学点数’。我开始计算花费时间所产生的‘教学点数’回报，并开始选择那些回报高的事情，而非我应该做的事情，例如和学生在一起。这么做，并非努力去获得更多的快乐，我也并不认为那些得到‘教学点数’高的事情会让我的教学更加高效，我只是被动地迎合学校这个衡量工具的评价，因此我能做的只是在这套游戏规则中做得最好（实际上，我承认我的确获得了很高的‘教学点数’）……”

类似的情况一定也在我们的工作中出现过！正如托马斯·乔纳森（H.Thomas Johnson）和安德鲁·布鲁明（Anders Broms）指出的：在大多数情况下，如果不是所有的话，在工作中你衡量什么，就会得到什么，也只会得到这些。对于那些没有或者无法被衡量的部分，很遗憾，你无法得到什么。”

对于一个 100 人的团队，其目标和绩效一定是需要被衡量的，这样才能让团队成员朝着一个方向交付高绩效。但是，如何充分发挥 KPI 的作用，而避免陷入上面“唯 KPI 论”的陷阱中呢？这里要介绍一个有效的管理 2.0 工具：OKR（Objective and Key Results，目标和关键成果法）。

OKR 由英特尔公司最先发明和使用，但是被越来越多的人认识，更多地是在近年来被硅谷的互联网公司如谷歌、Uber、Twitter 等大量使用之后。甚至有人认为 OKR 可以替代 KPI 对组织、团队和员工进行目标设定。实际上，我们认为 OKR 更多地应该和 KPI 一同使用，才能组成一个闭环的 Smart KPI。为什么这么说呢？让我们先从 OKR 的定义说起。

OKR 的含义是“目标和关键成果”，简单地说，目标确定了前进的方向，而关键成果能告诉我们需要取得什么样的结果，才能确保达到了预定的目标。另一方面，KPI 能够对结果提供精确和定量的评估。因此，结合 OKR 和 KPI，就能够完整地涵盖目标、关键成果，以及结果评估这三个相辅相成的重要方面，从而形成一个目标设定、评估和持续提升的闭环结构。

具体来说，结合 OKR 和 KPI，可以按照如下步骤实施。

第 1 步：根据目标和验收标准制订 OKR（目标和关键成果）

在之前讲解技术领导力 4D 框架时，介绍了制定目标和确定验收标准，接下来就可以根据制定的目标设定 OKR 中的目标部分，再根据验收标准确定 OKR 中的关键成果部分。例如，对于一个客户服务团队，假设在 2018 年设定的目标是提升客户的满意度，在相应的验收标准中包含一条“在 2018 年由第三方进行的本行业客户满意度调查中得分进入行业前三”，就可以据此制订相应的 OKR，如下所示。

目标：提升“客户满意度”。

关键成果：

（1）降低对客户投或咨询的平均响应时间至 30 分钟以内。

（2）提升客户投诉或咨询的满意度到 85%以上。

（3）在年度由第三方进行的本行业客户满意度调查中得分进入行业前三。

由此可以看到，在制订“关键成果”时，需要同时考虑过程和最终结果两个维度。

第 2 步：为 OKR 中的关键成果制订 KPI

为“关键成果”制订 KPI，才能够确保其得到精确和及时评估。因为“你衡量什么，就会得到什么”，同时，由于在制订关键成果时已经考虑到了兼顾过程和最终结果，因此相应制订的 KPI 自然也包括了先导性指标和滞后性指标。同样回到上面客户服务团队的例子，我们可以为关键成果制订如下 KPI。

（1）关键成果 1：降低对客户投诉或咨询的平均响应时间至 30 分钟以内。

KPI：先导性指标——对客户投诉或咨询的平均响应时间。

（2）关键成果 2：提升客户投诉或咨询的满意度到 85%以上。

KPI：先导性指标——客户投诉或咨询的满意度。

（3）关键成果 3：在由第三方进行的本行业年度客户满意度调查中得分进入行业前三。

KPI：滞后性指标——第三方行业客户满意度评测得分。

第 3 步：在监测先导性指标过程中巧妙利用 OKR 进行纠偏

监测先导性指标，可以快速和及时地对目标达成情况进行反馈。如果监测到先导性指标出现偏差，则可以再一次制订子 OKR 快速进行纠偏。同样，在上面的例子中，如果先导性指标“对客户投诉或咨询的平均响应时间”在某个时间段的监测中超过了 40 分钟（目标是 30 分钟），则可以立即制订一个子 OKR。

目标：提升客户投诉/咨询响应速度。

关键成果：将对客户投诉或咨询的平均响应时间从 40 分钟减少到 28 分钟。

基于这个子 OKR，团队可以再做出更为详细的行动计划并及时采取行动，直到取得关键成果且相应的先导性指标恢复到目标范围内。

当团队管理者综合使用上述关于团队愿景，以及 OKR 的管理 2.0 工具时，将会得到的结果是：根据团队的讨论达成一致的愿景或目标，以及相应的验收标准制订的 OKR（目标和关键成果）。

OKR 中的关键成果能够同时覆盖过程和最终结果两个维度，同时每一个关键成果都有相应的 KPI。由于在关键成果制订时考虑了过程和最终结果两个维度，因此相应的 KPI 既包括先导性指标，也包括滞后性指标。

并且，当 KPI 监测出现偏差时，也能够针对出现的偏差再次制订子 OKR 快速进行纠偏，从而形成“OKR 目标设定→KPI 监控→子 OKR 纠偏→目标达成”的闭环。

8.9 如何对上管理

张强（化名）是一家电商公司的项目主管，最近因为项目延期万分苦恼。某个周末的下午，张强约了曾经带过他的项目经理小宇（化名）并向小宇请教。因为工作出色，小宇现在已经是公司的技术总监，在项目管理方面颇有经验。

在短暂的寒暄之后，张强就开始描述自己最近的困扰：张强最近在负责公司的一个核心项目，但是老板在项目实施过程中修改了项目方案，在这之后的一个月里，为了追上项目工期，张强组织项目组的同事们夜以继日地工作，但因为方案过于复杂，项目还是延期了。而张强也因为没有及时将风险上报给老板而受到了严厉处理，同事们也因为没有正向输出不停责备张强。

听完张强的描述以后，小宇与张强进行了如下对话。

小宇：老板提出修改方案的时候，你是否真正理解到他的意思？若没有理解，你是否追问他直到得到一个清晰的答案？

张强：没有……

小宇：那你觉得老板当时提出的方案有问题吗？

张强：没有细想……

小宇：你在实施过程中发现风险发生时，有没有及时向老板汇报？或者向老板申请增加工时，或商讨替换方案？

张强：没有，我害怕受到责备！

……

不难想到，在上面这个案例中，小宇听完张强的回答以后一定面露难色。原因很简单：作为管理中的佼佼者，小宇非常清楚在张强的惨状就是典型的“对上预期管理不当”。造成这种惨状的主要原因如下：

◎ 对需求理解不透彻，没有做出正确的判断；
◎ 当风险发生时没有及时沟通。

那么，作为管理者，我们应该以什么样的方式做好对上管理呢？下面一起来探讨这个问题。

8.9.1　做好对上的预期管理

要做好对上的预期管理，我们通常需要在如下两方面下力气。

1. 彻底理解需求

所谓“知之为知之，不知为不知，是知也”，很多人在跟老板沟通时，为了保持在老板心目中的聪慧形象而放弃提问。然而，因为认知、信息来源和立场不同，老板在需求描述时给出的信息，不一定是所有人都能完全理解的，所以要敢于提问，理解、确认直至彻底理解需求，千万别带着疑问上路。

2. 学会站在老板的角度理解 KPI，求同存异

很多人面对老板提出的要求，会一味顺应。但老板不是神，很多时候也不能把问题完全想清楚。我们要做的，是站在老板的角度理解 KPI，若发现老板的方案有问题，则请及时把他从神坛拉回，这就是求同存异。在 KPI 上和老板保持统一，但是带着脑子去听方案，你的老板需要的也是一个能理解清楚 KPI 及提出自己见解的人，而不是一个只会“拍手掌”的人。

8.9.2　及时汇报

工作汇报往往被视为工作能力的体现，甚至超越对员工工作能力的评估，很多人因为害怕出错或对领导存有个人意见而放弃汇报，让团队及自身变得非常糟糕。

要把工作汇报作为一个正常交流的过程，沟通双方就需要放下心中芥蒂，平等、客观、简单地交换信息，员工准确表达自己对工作的理解，让领导拥有做决策的依据。领导也希望员工能及时汇报，以更多地掌握关键信息作为决策依据。一言以蔽之：及时汇报传递信息、发现工作中的问题不光是为了减少公司损失，也是为了让领导更了解你，因为领导通常会思考如下问题。

（1）你的价值在哪里？研发工作者很难直接跟公司业务指标（例如营收，利润等）产生联系，所以其价值往往很难被定义，因此应当不停地思考且经常就思考结果与领导进行沟通，及时发现问题和风险并提出解决方案，通过这种工作上的沟通和最终的落地去展现自己的能力，用行动去做自我价值画像，逐渐回答领导心中的问题。

（2）在团队中的角色定位是否清晰？因为团队也背负着自己的 KPI，所以在工作中遇到重大决策时，不要轻易、盲目地执行。毕竟你也只是负责团队中的一部分工作，很难了解到你的决策对全局的影响。所以，作为团队的一部分，我们需要及时将问题及自己的想法与领导进行沟通，让其掌握充分的信息，最终才能做出正确的决策。

刚才讲到及时汇报的重要性，这里再说说如何做好汇报。

（1）认真聆听。因为汇报也是一个沟通的过程，所以一定要认仔细聆听且理解领导的意思。

（2）简单扼要地表达。尽量用简洁的语言将自己需要汇报的内容表达出来，这样有利于领导获取信息，所以建议在汇报前罗列好汇报清单，而不是在汇报中组织汇报清单。

（3）客观描述，不要避重就轻。很多时候，大家往往会因为害怕责备而放弃客观描述，但问题永远是客观存在的，迟早会爆发出来，甚至以一种糟糕的方式爆发，等到那时就百口莫辩了，而且很难挽回领导对你的正向认知。

有了上面这些知识“武装自己”，你就会发现对上管理也并非难事。只要不断提醒自己，清晰地了解目标，做好预期管理，学会更高格局地思考并且及时沟通，你就能在对上管理方面收获更好的结果。